# EvalML Documentation

*Release 0.74.0*

**Alteryx, Inc.**

**Apr 19, 2023**

# CONTENTS


1 Install 3

2 Start 7

3 Tutorials 17

4 User Guide 51

5 API Reference 209

6 Release Notes 1923

Python Module Index 1995

Index 1999

EvalML is an AutoML library that builds, optimizes, and evaluates machine learning pipelines using domain-specific objective functions.

Combined with Featuretools and Compose, EvalML can be used to create end-to-end supervised machine learning solutions.

CHAPTER

ONE

# INSTALL

EvalML is available for Python 3.8 and 3.9. It can be installed from pypi, conda-forge, or from source.

To install EvalML on your platform, run one of the following commands:

```
$ pip install evalml
```

```
$ conda install -c conda-forge evalml
```

```
# See the EvalML with core dependencies only section
$ pip install evalml --no-dependencies
$ pip install -r core-requirements.txt
```

```
# See the EvalML with core dependencies only section
$ conda install -c conda-forge evalml-core
```

## 1.1 EvalML with core dependencies only

EvalML includes several optional dependencies. The `xgboost` and `catboost` packages support pipelines built around those modeling libraries. The `plotly` and `ipywidgets` packages support plotting functionality in automl searches. These dependencies are recommended, and are included with EvalML by default but are not required in order to install and use EvalML.

EvalML's core dependencies are listed in `core-requirements.txt` in the source code, while the default collection of requirements is specified in `pyproject.toml`'s `dependencies`.

To install EvalML with only the core-required dependencies with pypi, first download the EvalML source from pypi or github to access the requirements files before running the following command.

```
$ pip install evalml --no-dependencies
$ pip install -r core-requirements.txt
```

```
$ conda install -c conda-forge evalml-core
```

## 1.2 Add-ons

EvalML allows users to install add-ons individually or all at once:

- **Update Checker**: Receive automatic notifications of new EvalML releases
- **Time Series**: Use EvalML with Facebook's Prophet library for time series support.

```
$ pip install evalml[complete]
```

```
$ pip install evalml[prophet]
```

```
$ pip install evalml[updater]
```

```
$ conda install -c conda-forge alteryx-open-src-update-checker
```

## 1.3 Time Series support with Facebook's Prophet

To support the `Prophet` time series estimator, be sure to install it as an extra requirement. Please note that this may take a few minutes.

```
pip install evalml[prophet]
```

Another option for installing Prophet with CmdStan as a backend is to use `make installdeps-prophet`.

## 1.4 Windows Additional Requirements & Troubleshooting

If you are using `pip` to install EvalML on Windows, it is recommended you first install the following packages using conda:

- `numba` (needed for `shap` and prediction explanations). Install with `conda install -c conda-forge numba`
- `graphviz` if you're using EvalML's plotting utilities. Install with `conda install -c conda-forge python-graphviz`

The XGBoost library may not be pip-installable in some Windows environments. If you are encountering installation issues, please try installing XGBoost from Github before installing EvalML or install evalml with conda.

## 1.5 Mac Additional Requirements & Troubleshooting

In order to run on Mac, LightGBM requires the `OpenMP` library to be installed, which can be done with HomeBrew by running:

```
brew install libomp
```

Additionally, `graphviz` can be installed by running:

```
brew install graphviz
```

### 1.5.1 Installing EvalML on an M1 Mac

Not all of EvalML's dependencies support Apple's new M1 chip. For this reason, `pip` or `conda` installing EvalML will fail. The core set of EvalML dependencies can be installed in the M1 chip, so we recommend you install EvalML with core dependencies.

Alternatively, there is experimental support for M1 chips with the Rosetta terminal. After setting up a Rosetta terminal, you should be able to `pip` or `conda` install EvalML.

For Docker fans, an included `Dockerfile.arm` can be built and run to provide an environment for testing. Details are included within.

CHAPTER

TWO

# START

In this guide, we'll show how you can use EvalML to automatically find the best pipeline for predicting whether or not a credit card transaction is fradulent. Along the way, we'll highlight EvalML's built-in tools and features for understanding and interacting with the search process.

[1]:
```
import evalml
from evalml import AutoMLSearch
from evalml.utils import infer_feature_types
```

First, we load in the features and outcomes we want to use to train our model.

[2]:
```
X, y = evalml.demos.load_fraud(n_rows=250)
```

```
             Number of Features
Boolean                       1
Categorical                   6
Numeric                       5

Number of training examples: 250
Targets
False    88.40%
True     11.60%
Name: fraud, dtype: object
```

First, we will clean the data. Since EvalML accepts a pandas input, it can run type inference on this data directly. Since we'd like to change the types inferred by EvalML, we can use the `infer_feature_types` utility method. Here's what we're going to do with the following dataset:

- Reformat the `expiration_date` column so it reflects a more familiar date format.
- Cast the `lat` and `lng` columns from float to str.
- Use `infer_feature_types` to specify what types certain columns should be. For example, to avoid having the `provider` column be inferred as natural language text, we have specified it as a categorical column instead.

The `infer_feature_types` utility method takes a pandas or numpy input and converts it to a pandas dataframe with a Woodwork accessor, providing us with flexibility to cast the data as necessary.

[3]:
```
X.ww["expiration_date"] = X["expiration_date"].apply(
    lambda x: "20{}-01-{}".format(x.split("/")[1], x.split("/")[0])
)
X = infer_feature_types(
    X,
    feature_types={
```

(continues on next page)

(continued from previous page)

```
        "store_id": "categorical",
        "expiration_date": "datetime",
        "lat": "categorical",
        "lng": "categorical",
        "provider": "categorical",
    },
)
X.ww
```

[3]:
```
                   Physical Type Logical Type Semantic Tag(s)
Column
card_id                    int64      Integer     ['numeric']
store_id                   int64      Integer     ['numeric']
datetime          datetime64[ns]     Datetime              []
amount                     int64      Integer     ['numeric']
currency                  string      Unknown              []
customer_present            bool      Boolean              []
expiration_date   datetime64[ns]     Datetime              []
provider                 category  Categorical    ['category']
lat                       float64       Double     ['numeric']
lng                       float64       Double     ['numeric']
region                   category  Categorical    ['category']
country                  category  Categorical    ['category']
```

In order to validate the results of the pipeline creation and optimization process, we will save some of our data as a holdout set.

[4]:
```
X_train, X_holdout, y_train, y_holdout = evalml.preprocessing.split_data(
    X, y, problem_type="binary", test_size=0.2
)
```

**Note:** To provide data to EvalML, it is recommended that you initialize a woodwork accessor so that you control how EvalML will treat each feature, such as as a numeric feature, a categorical feature, a text feature or other type of feature. Consult the the Woodwork project for help on how to do this. Here, `split_data()` returns dataframes with woodwork accessors.

EvalML has many options to configure the pipeline search. At the minimum, we need to define an objective function. For simplicity, we will use the F1 score in this example. However, the real power of EvalML is in using domain-specific *objective functions* or building your own.

Below EvalML utilizes Bayesian optimization (EvalML's default optimizer) to search and find the best pipeline defined by the given objective.

EvalML provides a number of parameters to control the search process. `max_batches` is one of the parameters which controls the stopping criterion for the AutoML search. It indicates the maximum number of rounds of AutoML to evaluate, where each round may train and score a variable number of pipelines. In this example, `max_batches` is set to 1.

** Graphing methods, like AutoMLSearch, on Jupyter Notebook and Jupyter Lab require ipywidgets to be installed.

** If graphing on Jupyter Lab, jupyterlab-plotly required. To download this, make sure you have npm installed.

[5]:
```
automl = AutoMLSearch(
    X_train=X_train,
    y_train=y_train,
```

(continues on next page)

(continued from previous page)

```
    problem_type="binary",
    objective="f1",
    max_batches=3,
    verbose=True,
)
```

```
AutoMLSearch will use mean CV score to rank pipelines.
Removing columns ['currency'] because they are of 'Unknown' type
```

When we call `search()`, the search for the best pipeline will begin. There is no need to wrangle with missing data or categorical variables as EvalML includes various preprocessing steps (like imputation, one-hot encoding, feature selection) to ensure you're getting the best results. As long as your data is in a single table, EvalML can handle it. If not, you can reduce your data to a single table by utilizing Featuretools and its Entity Sets.

You can find more information on pipeline components and how to integrate your own custom pipelines into EvalML *here*.

[6]:
```
automl.search(interactive_plot=False)
```

```
*****************************
* Beginning pipeline search *
*****************************

Optimizing for F1.
Greater score is better.

Using SequentialEngine to train and score pipelines.
Searching up to 3 batches for a total of None pipelines.
Allowed model families:

Evaluating Baseline Pipeline: Mode Baseline Binary Classification Pipeline
Mode Baseline Binary Classification Pipeline:
        Starting cross validation
        Finished cross validation - mean F1: 0.000

*****************************
* Evaluating Batch Number 1 *
*****************************

Random Forest Classifier w/ Label Encoder + Drop Columns Transformer + DateTime␣
↪Featurizer + Imputer + One Hot Encoder + Oversampler:
        Starting cross validation
        Finished cross validation - mean F1: 0.558

*****************************
* Evaluating Batch Number 2 *
*****************************

Random Forest Classifier w/ Label Encoder + Drop Columns Transformer + DateTime␣
↪Featurizer + Imputer + One Hot Encoder + Oversampler + RF Classifier Select From Model:
        Starting cross validation
        Finished cross validation - mean F1: 0.663
```

(continues on next page)

(continued from previous page)

```
*****************************
* Evaluating Batch Number 3 *
*****************************

Decision Tree Classifier w/ Label Encoder + Select Columns By Type Transformer + Label␣
↪Encoder + Drop Columns Transformer + DateTime Featurizer + Imputer + Select Columns␣
↪Transformer + Select Columns Transformer + Label Encoder + Imputer + One Hot Encoder +␣
↪Oversampler:
        Starting cross validation
        Finished cross validation - mean F1: 0.289
LightGBM Classifier w/ Label Encoder + Select Columns By Type Transformer + Label␣
↪Encoder + Drop Columns Transformer + DateTime Featurizer + Imputer + Select Columns␣
↪Transformer + Select Columns Transformer + Label Encoder + Imputer + One Hot Encoder +␣
↪Oversampler:
        Starting cross validation
        Finished cross validation - mean F1: 0.589
Extra Trees Classifier w/ Label Encoder + Select Columns By Type Transformer + Label␣
↪Encoder + Drop Columns Transformer + DateTime Featurizer + Imputer + Select Columns␣
↪Transformer + Select Columns Transformer + Label Encoder + Imputer + One Hot Encoder +␣
↪Oversampler:
        Starting cross validation
        Finished cross validation - mean F1: 0.376
Elastic Net Classifier w/ Label Encoder + Select Columns By Type Transformer + Label␣
↪Encoder + Drop Columns Transformer + DateTime Featurizer + Imputer + Standard Scaler +␣
↪Select Columns Transformer + Select Columns Transformer + Label Encoder + Imputer +␣
↪One Hot Encoder + Standard Scaler + Oversampler:
        Starting cross validation
        Finished cross validation - mean F1: 0.395
CatBoost Classifier w/ Label Encoder + Select Columns By Type Transformer + Label␣
↪Encoder + Drop Columns Transformer + DateTime Featurizer + Imputer + Select Columns␣
↪Transformer + Select Columns Transformer + Label Encoder + Imputer + Oversampler:
        Starting cross validation
        Finished cross validation - mean F1: 0.730
XGBoost Classifier w/ Label Encoder + Select Columns By Type Transformer + Label Encoder␣
↪+ Drop Columns Transformer + DateTime Featurizer + Imputer + Select Columns␣
↪Transformer + Select Columns Transformer + Label Encoder + Imputer + One Hot Encoder +␣
↪Oversampler:
        Starting cross validation
        Finished cross validation - mean F1: 0.690
Logistic Regression Classifier w/ Label Encoder + Select Columns By Type Transformer +␣
↪Label Encoder + Drop Columns Transformer + DateTime Featurizer + Imputer + Standard␣
↪Scaler + Select Columns Transformer + Select Columns Transformer + Label Encoder +␣
↪Imputer + One Hot Encoder + Standard Scaler + Oversampler:
        Starting cross validation
        Finished cross validation - mean F1: 0.231

Search finished after 00:28
Best pipeline: CatBoost Classifier w/ Label Encoder + Select Columns By Type Transformer␣
↪+ Label Encoder + Drop Columns Transformer + DateTime Featurizer + Imputer + Select␣
↪Columns Transformer + Select Columns Transformer + Label Encoder + Imputer +␣
↪Oversampler
Best pipeline F1: 0.730159
```

Data type cannot be displayed: application/vnd.plotly.v1+json, text/html

[6]:
```
{1: {'Random Forest Classifier w/ Label Encoder + Drop Columns Transformer + DateTime␣
↪Featurizer + Imputer + One Hot Encoder + Oversampler': '00:02',
  'Total time of batch': '00:03'},
 2: {'Random Forest Classifier w/ Label Encoder + Drop Columns Transformer + DateTime␣
↪Featurizer + Imputer + One Hot Encoder + Oversampler + RF Classifier Select From Model
↪': '00:03',
  'Total time of batch': '00:03'},
 3: {'Decision Tree Classifier w/ Label Encoder + Select Columns By Type Transformer +␣
↪Label Encoder + Drop Columns Transformer + DateTime Featurizer + Imputer + Select␣
↪Columns Transformer + Select Columns Transformer + Label Encoder + Imputer + One Hot␣
↪Encoder + Oversampler': '00:02',
  'LightGBM Classifier w/ Label Encoder + Select Columns By Type Transformer + Label␣
↪Encoder + Drop Columns Transformer + DateTime Featurizer + Imputer + Select Columns␣
↪Transformer + Select Columns Transformer + Label Encoder + Imputer + One Hot Encoder +␣
↪Oversampler': '00:02',
  'Extra Trees Classifier w/ Label Encoder + Select Columns By Type Transformer + Label␣
↪Encoder + Drop Columns Transformer + DateTime Featurizer + Imputer + Select Columns␣
↪Transformer + Select Columns Transformer + Label Encoder + Imputer + One Hot Encoder +␣
↪Oversampler': '00:03',
  'Elastic Net Classifier w/ Label Encoder + Select Columns By Type Transformer + Label␣
↪Encoder + Drop Columns Transformer + DateTime Featurizer + Imputer + Standard Scaler +␣
↪Select Columns Transformer + Select Columns Transformer + Label Encoder + Imputer +␣
↪One Hot Encoder + Standard Scaler + Oversampler': '00:03',
  'CatBoost Classifier w/ Label Encoder + Select Columns By Type Transformer + Label␣
↪Encoder + Drop Columns Transformer + DateTime Featurizer + Imputer + Select Columns␣
↪Transformer + Select Columns Transformer + Label Encoder + Imputer + Oversampler': '00:
↪02',
  'XGBoost Classifier w/ Label Encoder + Select Columns By Type Transformer + Label␣
↪Encoder + Drop Columns Transformer + DateTime Featurizer + Imputer + Select Columns␣
↪Transformer + Select Columns Transformer + Label Encoder + Imputer + One Hot Encoder +␣
↪Oversampler': '00:02',
  'Logistic Regression Classifier w/ Label Encoder + Select Columns By Type Transformer␣
↪+ Label Encoder + Drop Columns Transformer + DateTime Featurizer + Imputer + Standard␣
↪Scaler + Select Columns Transformer + Select Columns Transformer + Label Encoder +␣
↪Imputer + One Hot Encoder + Standard Scaler + Oversampler': '00:04',
  'Total time of batch': '00:21'}}
```

If you would like to suppress stdout output, set `verbose=False`. This is also the default behavior for `AutoMLSearch` if `verbose` is not specified.

Also, if you would like to see the interactive plot update dynamically over time as the search progresses, either remove the parameter or set `interactive_plot=True`. This is the default setting for `search()` if `interactive_plot` is not specified (it is set to False here due to documentation workaround).

[7]:
```
automl = AutoMLSearch(
    X_train=X_train,
    y_train=y_train,
    problem_type="binary",
    objective="f1",
```

(continues on next page)

(continued from previous page)

```
    max_batches=3,
    verbose=False,
)
automl.search()
```

[7]:
```
{1: {'Random Forest Classifier w/ Label Encoder + Drop Columns Transformer + DateTime␣
↪Featurizer + Imputer + One Hot Encoder + Oversampler': '00:02',
  'Total time of batch': '00:02'},
 2: {'Random Forest Classifier w/ Label Encoder + Drop Columns Transformer + DateTime␣
↪Featurizer + Imputer + One Hot Encoder + Oversampler + RF Classifier Select From Model
↪': '00:03',
  'Total time of batch': '00:03'},
 3: {'Decision Tree Classifier w/ Label Encoder + Select Columns By Type Transformer +␣
↪Label Encoder + Drop Columns Transformer + DateTime Featurizer + Imputer + Select␣
↪Columns Transformer + Select Columns Transformer + Label Encoder + Imputer + One Hot␣
↪Encoder + Oversampler': '00:02',
  'LightGBM Classifier w/ Label Encoder + Select Columns By Type Transformer + Label␣
↪Encoder + Drop Columns Transformer + DateTime Featurizer + Imputer + Select Columns␣
↪Transformer + Select Columns Transformer + Label Encoder + Imputer + One Hot Encoder +␣
↪Oversampler': '00:02',
  'Extra Trees Classifier w/ Label Encoder + Select Columns By Type Transformer + Label␣
↪Encoder + Drop Columns Transformer + DateTime Featurizer + Imputer + Select Columns␣
↪Transformer + Select Columns Transformer + Label Encoder + Imputer + One Hot Encoder +␣
↪Oversampler': '00:03',
  'Elastic Net Classifier w/ Label Encoder + Select Columns By Type Transformer + Label␣
↪Encoder + Drop Columns Transformer + DateTime Featurizer + Imputer + Standard Scaler +␣
↪Select Columns Transformer + Select Columns Transformer + Label Encoder + Imputer +␣
↪One Hot Encoder + Standard Scaler + Oversampler': '00:03',
  'CatBoost Classifier w/ Label Encoder + Select Columns By Type Transformer + Label␣
↪Encoder + Drop Columns Transformer + DateTime Featurizer + Imputer + Select Columns␣
↪Transformer + Select Columns Transformer + Label Encoder + Imputer + Oversampler': '00:
↪02',
  'XGBoost Classifier w/ Label Encoder + Select Columns By Type Transformer + Label␣
↪Encoder + Drop Columns Transformer + DateTime Featurizer + Imputer + Select Columns␣
↪Transformer + Select Columns Transformer + Label Encoder + Imputer + One Hot Encoder +␣
↪Oversampler': '00:02',
  'Logistic Regression Classifier w/ Label Encoder + Select Columns By Type Transformer␣
↪+ Label Encoder + Drop Columns Transformer + DateTime Featurizer + Imputer + Standard␣
↪Scaler + Select Columns Transformer + Select Columns Transformer + Label Encoder +␣
↪Imputer + One Hot Encoder + Standard Scaler + Oversampler': '00:02',
  'Total time of batch': '00:19'}}
```

We also provide a standalone `search` *method* which does all of the above in a single line, and returns the `AutoMLSearch` instance and data check results. If there were data check errors, AutoML will not be run and no `AutoMLSearch` instance will be returned.

After the search is finished we can view all of the pipelines searched, ranked by score. Internally, EvalML performs cross validation to score the pipelines. If it notices a high variance across cross validation folds, it will warn you. EvalML also provides additional *data checks* to analyze your data to assist you in producing the best performing pipeline.

[8]:
```
automl.rankings
```

[8]:
```
   id                                     pipeline_name  search_order  \
0   7  CatBoost Classifier w/ Label Encoder + Select ...             7
```

(continues on next page)

(continued from previous page)

```
1   8  XGBoost Classifier w/ Label Encoder + Select C...             8
2   2  Random Forest Classifier w/ Label Encoder + Dr...             2
3   4  LightGBM Classifier w/ Label Encoder + Select ...             4
4   1  Random Forest Classifier w/ Label Encoder + Dr...             1
5   6  Elastic Net Classifier w/ Label Encoder + Sele...             6
6   5  Extra Trees Classifier w/ Label Encoder + Sele...             5
7   3  Decision Tree Classifier w/ Label Encoder + Se...             3
8   9  Logistic Regression Classifier w/ Label Encode...             9
9   0       Mode Baseline Binary Classification Pipeline             0

   ranking_score  mean_cv_score  standard_deviation_cv_score  \
0       0.730159       0.730159                     0.199679
1       0.689744       0.689744                     0.165041
2       0.663337       0.663337                     0.263244
3       0.588889       0.588889                     0.083887
4       0.558405       0.558405                     0.182781
5       0.395153       0.395153                     0.183837
6       0.376068       0.376068                     0.074019
7       0.289001       0.289001                     0.103728
8       0.231260       0.231260                     0.035912
9       0.000000       0.000000                     0.000000

   percent_better_than_baseline  high_variance_cv  \
0                     73.015873             False
1                     68.974359             False
2                     66.333666             False
3                     58.888889             False
4                     55.840456             False
5                     39.515251             False
6                     37.606838             False
7                     28.900112             False
8                     23.125997             False
9                      0.000000             False

                                        parameters
0  {'Label Encoder': {'positive_label': None}, 'N...
1  {'Label Encoder': {'positive_label': None}, 'N...
2  {'Label Encoder': {'positive_label': None}, 'D...
3  {'Label Encoder': {'positive_label': None}, 'N...
4  {'Label Encoder': {'positive_label': None}, 'D...
5  {'Label Encoder': {'positive_label': None}, 'N...
6  {'Label Encoder': {'positive_label': None}, 'N...
7  {'Label Encoder': {'positive_label': None}, 'N...
8  {'Label Encoder': {'positive_label': None}, 'N...
9  {'Label Encoder': {'positive_label': None}, 'B...
```

If we are interested in see more details about the pipeline, we can view a summary description using the `id` from the rankings table:

[9]:
```
automl.describe_pipeline(3)
```

```
*****************************************************************************************************
```

(continues on next page)

(continued from previous page)

```
* Decision Tree Classifier w/ Label Encoder + Select Columns By Type Transformer + Label␣
↪Encoder + Drop Columns Transformer + DateTime Featurizer + Imputer + Select Columns␣
↪Transformer + Select Columns Transformer + Label Encoder + Imputer + One Hot Encoder +␣
↪Oversampler *
*****************************************************************************************************************

Problem Type: binary
Model Family: Decision Tree

Pipeline Steps
==============
1. Label Encoder
         * positive_label : None
2. Select Columns By Type Transformer
         * column_types : ['category', 'EmailAddress', 'URL']
         * exclude : True
3. Label Encoder
         * positive_label : None
4. Drop Columns Transformer
         * columns : ['currency']
5. DateTime Featurizer
         * features_to_extract : ['year', 'month', 'day_of_week', 'hour']
         * encode_as_categories : False
         * time_index : None
6. Imputer
         * categorical_impute_strategy : most_frequent
         * numeric_impute_strategy : mean
         * boolean_impute_strategy : most_frequent
         * categorical_fill_value : None
         * numeric_fill_value : None
         * boolean_fill_value : None
7. Select Columns Transformer
         * columns : ['card_id', 'store_id', 'amount', 'customer_present', 'lat', 'lng',
↪'datetime_month', 'datetime_day_of_week', 'datetime_hour', 'expiration_date_year',
↪'expiration_date_day_of_week']
8. Select Columns Transformer
         * columns : ['provider', 'region', 'country']
9. Label Encoder
         * positive_label : None
10. Imputer
         * categorical_impute_strategy : most_frequent
         * numeric_impute_strategy : mean
         * boolean_impute_strategy : most_frequent
         * categorical_fill_value : None
         * numeric_fill_value : None
         * boolean_fill_value : None
11. One Hot Encoder
         * top_n : 10
         * features_to_encode : None
         * categories : None
         * drop : if_binary
         * handle_unknown : ignore
```

(continues on next page)

(continued from previous page)

```
        * handle_missing : error
12. Oversampler
        * sampling_ratio : 0.25
        * k_neighbors_default : 5
        * n_jobs : -1
        * sampling_ratio_dict : None
        * categorical_features : [3, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23,
↪ 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40]
        * k_neighbors : 5
13. Decision Tree Classifier
        * criterion : gini
        * max_features : auto
        * max_depth : 6
        * min_samples_split : 2
        * min_weight_fraction_leaf : 0.0

Training
========
Training for binary problems.
Objective to optimize binary classification pipeline thresholds for: <evalml.objectives.
↪standard_metrics.F1 object at 0x7fbd28d4b0a0>
Total training time (including CV): 2.4 seconds

Cross Validation
----------------
               F1  MCC Binary  Log Loss Binary   Gini   AUC  Precision  Balanced␣
↪Accuracy Binary  Accuracy Binary # Training # Validation
0           0.182       0.037            4.003 -0.036 0.482      0.143                ␣
↪  0.523            0.731        133           67
1           0.296       0.177            2.350  0.153 0.576      0.211                ␣
↪  0.623            0.716        133           67
2           0.389       0.389            5.623  0.608 0.804      0.241                ␣
↪  0.814            0.667        134           66
mean        0.289       0.201            3.992  0.241 0.621      0.198                ␣
↪  0.653            0.705          -            -
std         0.104       0.177            1.637  0.331 0.165      0.050                ␣
↪  0.147            0.034          -            -
coef of var 0.359       0.881            0.410  1.371 0.267      0.254                ␣
↪  0.226            0.048          -            -
```

We can also view the pipeline parameters directly:

[10]:
```
pipeline = automl.get_pipeline(3)
print(pipeline.parameters)
```

```
{'Label Encoder': {'positive_label': None}, 'Numeric Pipeline - Select Columns By Type␣
↪Transformer': {'column_types': ['category', 'EmailAddress', 'URL'], 'exclude': True},
↪'Numeric Pipeline - Label Encoder': {'positive_label': None}, 'Numeric Pipeline - Drop␣
↪Columns Transformer': {'columns': ['currency']}, 'Numeric Pipeline - DateTime␣
↪Featurizer': {'features_to_extract': ['year', 'month', 'day_of_week', 'hour'], 'encode_
↪as_categories': False, 'time_index': None}, 'Numeric Pipeline - Imputer': {
↪'categorical_impute_strategy': 'most_frequent', 'numeric_impute_strategy': 'mean',
↪'boolean_impute_strategy': 'most_frequent', 'categorical_fill_value': None, 'numeric_
↪fill_value': None, 'boolean_fill_value': None}, 'Numeric Pipeline - Select Columns␣
↪Transformer': {'columns': ['card_id', 'store_id', 'amount', 'customer_present', 'lat',
↪'lng', 'datetime_month', 'datetime_day_of_week', 'datetime_hour', 'expiration_date_year
↪', 'expiration_date_day_of_week']}, 'Categorical Pipeline - Select Columns Transformer
↪': {'columns': ['provider', 'region', 'country']}, 'Categorical Pipeline - Label␣
↪Encoder': {'positive_label': None}, 'Categorical Pipeline - Imputer': {'categorical_
↪impute_strategy': 'most_frequent', 'numeric_impute_strategy': 'mean', 'boolean_impute
```

(continues on next page)

(continued from previous page)

We can now select the best pipeline and score it on our holdout data:

[11]:
```
pipeline = automl.best_pipeline
pipeline.score(X_holdout, y_holdout, ["f1"])
```

[11]:
```
OrderedDict([('F1', 0.8)])
```

We can also visualize the structure of the components contained by the pipeline:

[12]:
```
pipeline.graph()
```

[12]:

CHAPTER

THREE

# TUTORIALS

Below are examples of how to apply EvalML to a variety of problems:

## 3.1 Building a Fraud Prediction Model with EvalML

In this demo, we will build an optimized fraud prediction model using EvalML. To optimize the pipeline, we will set up an objective function to minimize the percentage of total transaction value lost to fraud. At the end of this demo, we also show you how introducing the right objective during the training results in a much better than using a generic machine learning metric like AUC.

[1]:
```
import evalml
from evalml import AutoMLSearch
from evalml.objectives import FraudCost
```

### 3.1.1 Configure "Cost of Fraud"

To optimize the pipelines toward the specific business needs of this model, we can set our own assumptions for the cost of fraud. These parameters are

- `retry_percentage` - what percentage of customers will retry a transaction if it is declined?
- `interchange_fee` - how much of each successful transaction do you collect?
- `fraud_payout_percentage` - the percentage of fraud will you be unable to collect
- `amount_col` - the column in the data the represents the transaction amount

Using these parameters, EvalML determines attempt to build a pipeline that will minimize the financial loss due to fraud.

[2]:
```
fraud_objective = FraudCost(
    retry_percentage=0.5,
    interchange_fee=0.02,
    fraud_payout_percentage=0.75,
    amount_col="amount",
)
```

### 3.1.2 Search for best pipeline

In order to validate the results of the pipeline creation and optimization process, we will save some of our data as the holdout set.

[3]:
```
X, y = evalml.demos.load_fraud(n_rows=5000)
```

```
             Number of Features
Boolean                       1
Categorical                   6
Numeric                       5

Number of training examples: 5000
Targets
False    86.20%
True     13.80%
Name: fraud, dtype: object
```

EvalML natively supports one-hot encoding. Here we keep 1 out of the 6 categorical columns to decrease computation time.

[4]:
```
cols_to_drop = ["datetime", "expiration_date", "country", "region", "provider"]
for col in cols_to_drop:
    X.ww.pop(col)

X_train, X_holdout, y_train, y_holdout = evalml.preprocessing.split_data(
    X, y, problem_type="binary", test_size=0.2, random_seed=0
)

X.ww
```

[4]:
```
                 Physical Type Logical Type Semantic Tag(s)
Column
card_id                  int64      Integer     ['numeric']
store_id                 int64      Integer     ['numeric']
amount                   int64      Integer     ['numeric']
currency              category  Categorical    ['category']
customer_present          bool      Boolean              []
lat                    float64       Double     ['numeric']
lng                    float64       Double     ['numeric']
```

Because the fraud labels are binary, we will use `AutoMLSearch(X_train=X_train, y_train=y_train, problem_type='binary')`. When we call `.search()`, the search for the best pipeline will begin.

[5]:
```
automl = AutoMLSearch(
    X_train=X_train,
    y_train=y_train,
    problem_type="binary",
    objective=fraud_objective,
    additional_objectives=["auc", "f1", "precision"],
    allowed_model_families=["random_forest", "linear_model"],
    max_batches=1,
    optimize_thresholds=True,
    verbose=True,
)
```

(continues on next page)

(continued from previous page)

```
automl.search(interactive_plot=False)
```

```
AutoMLSearch will use mean CV score to rank pipelines.

*****************************
* Beginning pipeline search *
*****************************

Optimizing for Fraud Cost.
Lower score is better.

Using SequentialEngine to train and score pipelines.
Searching up to 1 batches for a total of None pipelines.
Allowed model families:

Evaluating Baseline Pipeline: Mode Baseline Binary Classification Pipeline
Mode Baseline Binary Classification Pipeline:
        Starting cross validation
        Finished cross validation - mean Fraud Cost: 0.790

*****************************
* Evaluating Batch Number 1 *
*****************************

Random Forest Classifier w/ Label Encoder + Imputer + One Hot Encoder + Oversampler:
        Starting cross validation
        Finished cross validation - mean Fraud Cost: 0.009

Search finished after 00:04
Best pipeline: Random Forest Classifier w/ Label Encoder + Imputer + One Hot Encoder +␣
↪Oversampler
Best pipeline Fraud Cost: 0.008745
```

Data type cannot be displayed: application/vnd.plotly.v1+json, text/html

[5]:
```
{1: {'Random Forest Classifier w/ Label Encoder + Imputer + One Hot Encoder + Oversampler
↪': '00:03',
  'Total time of batch': '00:03'}}
```

### View rankings and select pipelines

Once the fitting process is done, we can see all of the pipelines that were searched, ranked by their score on the fraud detection objective we defined.

[6]:
```
automl.rankings
```

[6]:
```
   id                                     pipeline_name  search_order  \
0   1  Random Forest Classifier w/ Label Encoder + Im...             1
1   0       Mode Baseline Binary Classification Pipeline             0
```

(continues on next page)

(continued from previous page)

```
   ranking_score  mean_cv_score  standard_deviation_cv_score  \
0       0.008745       0.008745                     0.001207
1       0.789648       0.789648                     0.001136

   percent_better_than_baseline  high_variance_cv  \
0                     78.090334             False
1                      0.000000             False

                                          parameters
0  {'Label Encoder': {'positive_label': None}, 'I...
1  {'Label Encoder': {'positive_label': None}, 'B...
```

To select the best pipeline we can call `automl.best_pipeline`.

[7]:
```
best_pipeline = automl.best_pipeline
```

### Describe pipelines

We can get more details about any pipeline created during the search process, including how it performed on other objective functions, by calling the `describe_pipeline` method and passing the `id` of the pipeline of interest.

[8]:
```
automl.describe_pipeline(automl.rankings.iloc[1]["id"])
```

```
************************************************
* Mode Baseline Binary Classification Pipeline *
************************************************

Problem Type: binary
Model Family: Baseline

Pipeline Steps
==============
1. Label Encoder
         * positive_label : None
2. Baseline Classifier
         * strategy : mode

Training
========
Training for binary problems.
Objective to optimize binary classification pipeline thresholds for: <evalml.objectives.
↪fraud_cost.FraudCost object at 0x7fc3248f1730>
Total training time (including CV): 0.7 seconds

Cross Validation
----------------
             Fraud Cost   AUC    F1  Precision # Training # Validation
0                 0.791 0.500 0.000      0.000      2,666        1,334
1                 0.789 0.500 0.000      0.000      2,667        1,333
2                 0.789 0.500 0.000      0.000      2,667        1,333
```

(continues on next page)

(continued from previous page)

```
mean              0.790 0.500 0.000      0.000          -            -
std               0.001 0.000 0.000      0.000          -            -
coef of var       0.001 0.000   inf        inf          -            -
```

### 3.1.3 Evaluate on holdout data

Finally, since the best pipeline is already trained, we evaluate it on the holdout data.

Now, we can score the pipeline on the holdout data using both our fraud cost objective and the AUC (Area under the ROC Curve) objective.

[9]:
```
best_pipeline.score(X_holdout, y_holdout, objectives=["auc", fraud_objective])
```

[9]:
```
OrderedDict([('AUC', 0.8644456773933219),
             ('Fraud Cost', 0.009295721543224191)])
```

### 3.1.4 Why optimize for a problem-specific objective?

To demonstrate the importance of optimizing for the right objective, let's search for another pipeline using AUC, a common machine learning metric. After that, we will score the holdout data using the fraud cost objective to see how the best pipelines compare.

[10]:
```
automl_auc = AutoMLSearch(
    X_train=X_train,
    y_train=y_train,
    problem_type="binary",
    objective="auc",
    additional_objectives=["f1", "precision"],
    max_batches=1,
    allowed_model_families=["random_forest", "linear_model"],
    optimize_thresholds=True,
    verbose=True,
)

automl_auc.search(interactive_plot=False)
```

```
AutoMLSearch will use mean CV score to rank pipelines.

*****************************
* Beginning pipeline search *
*****************************

Optimizing for AUC.
Greater score is better.

Using SequentialEngine to train and score pipelines.
Searching up to 1 batches for a total of None pipelines.
Allowed model families:

Evaluating Baseline Pipeline: Mode Baseline Binary Classification Pipeline
Mode Baseline Binary Classification Pipeline:
```

(continues on next page)

(continued from previous page)

```
        Starting cross validation
        Finished cross validation - mean AUC: 0.500

*****************************
* Evaluating Batch Number 1 *
*****************************

Random Forest Classifier w/ Label Encoder + Imputer + One Hot Encoder + Oversampler:
        Starting cross validation
        Finished cross validation - mean AUC: 0.853

Search finished after 00:03
Best pipeline: Random Forest Classifier w/ Label Encoder + Imputer + One Hot Encoder +␣
↪Oversampler
Best pipeline AUC: 0.853164
```

Data type cannot be displayed: application/vnd.plotly.v1+json, text/html

[10]:
```
{1: {'Random Forest Classifier w/ Label Encoder + Imputer + One Hot Encoder + Oversampler
↪': '00:02',
  'Total time of batch': '00:02'}}
```

Like before, we can look at the rankings of all of the pipelines searched and pick the best pipeline.

[11]:
```
automl_auc.rankings
```

[11]:
```
   id                                     pipeline_name  search_order  \
0   1  Random Forest Classifier w/ Label Encoder + Im...             1
1   0       Mode Baseline Binary Classification Pipeline             0

   ranking_score  mean_cv_score  standard_deviation_cv_score  \
0       0.853164       0.853164                     0.010011
1       0.500000       0.500000                     0.000000

   percent_better_than_baseline  high_variance_cv  \
0                     35.316397             False
1                      0.000000             False

                                          parameters
0  {'Label Encoder': {'positive_label': None}, 'I...
1  {'Label Encoder': {'positive_label': None}, 'B...
```

[12]:
```
best_pipeline_auc = automl_auc.best_pipeline
```

[13]:
```
# get the fraud score on holdout data
best_pipeline_auc.score(X_holdout, y_holdout, objectives=["auc", fraud_objective])
```

[13]:
```
OrderedDict([('AUC', 0.8644456773933219), ('Fraud Cost', 0.02574778650753432)])
```

[14]:
```
# fraud score on fraud optimized again
best_pipeline.score(X_holdout, y_holdout, objectives=["auc", fraud_objective])
```

[14]:
```
OrderedDict([('AUC', 0.8644456773933219),
             ('Fraud Cost', 0.009295721543224191)])
```

When we optimize for AUC, we can see that the AUC score from this pipeline performs better compared to the AUC score from the pipeline optimized for fraud cost; however, the losses due to fraud are a much larger percentage of the total transaction amount when optimized for AUC and much smaller when optimized for fraud cost. As a result, we lose a noticable percentage of the total transaction amount by not optimizing for fraud cost specifically.

Optimizing for AUC does not take into account the user-specified `retry_percentage`, `interchange_fee`, `fraud_payout_percentage` values, which could explain the decrease in fraud performance. Thus, the best pipelines may produce the highest AUC but may not actually reduce the amount loss due to your specific type fraud.

This example highlights how performance in the real world can diverge greatly from machine learning metrics.

## 3.2 Building a Lead Scoring Model with EvalML

In this demo, we will build an optimized lead scoring model using EvalML. To optimize the pipeline, we will set up an objective function to maximize the revenue generated with true positives while taking into account the cost of false positives. At the end of this demo, we also show you how introducing the right objective during the training is significantly better than using a generic machine learning metric like AUC.

[1]:
```
import evalml
from evalml import AutoMLSearch
from evalml.objectives import LeadScoring
```

### 3.2.1 Configure LeadScoring

To optimize the pipelines toward the specific business needs of this model, you can set your own assumptions for how much value is gained through true positives and the cost associated with false positives. These parameters are

- `true_positive` - dollar amount to be gained with a successful lead
- `false_positive` - dollar amount to be lost with an unsuccessful lead

Using these parameters, EvalML builds a pileline that will maximize the amount of revenue per lead generated.

[2]:
```
lead_scoring_objective = LeadScoring(true_positives=25, false_positives=-5)
```

### 3.2.2 Dataset

We will be utilizing a dataset detailing a customer's job, country, state, zip, online action, the dollar amount of that action and whether they were a successful lead.

[3]:
```
from urllib.request import urlopen
import pandas as pd
import woodwork as ww

customers_data = urlopen(
    "https://featurelabs-static.s3.amazonaws.com/lead_scoring_ml_apps/customers.csv"
)
interactions_data = urlopen(
```

(continues on next page)

(continued from previous page)

```
    "https://featurelabs-static.s3.amazonaws.com/lead_scoring_ml_apps/interactions.csv"
)
leads_data = urlopen(
    "https://featurelabs-static.s3.amazonaws.com/lead_scoring_ml_apps/previous_leads.csv"
)
customers = pd.read_csv(customers_data)
interactions = pd.read_csv(interactions_data)
leads = pd.read_csv(leads_data)

X = customers.merge(interactions, on="customer_id").merge(leads, on="customer_id")
y = X["label"]
X = X.drop(
    [
        "customer_id",
        "date_registered",
        "birthday",
        "phone",
        "email",
        "owner",
        "company",
        "id",
        "time_x",
        "session",
        "referrer",
        "time_y",
        "label",
        "country",
    ],
    axis=1,
)
display(X.head())
```

```
                      job state      zip     action  amount
0        Engineer, mining    NY  60091.0  page_view     NaN
1  Psychologist, forensic    CA      NaN   purchase  135.23
2  Psychologist, forensic    CA      NaN  page_view     NaN
3          Air cabin crew   NaN  60091.0   download     NaN
4          Air cabin crew   NaN  60091.0  page_view     NaN
```

We will convert our data into Woodwork data structures. Doing so enables us to have more control over the types passed to and inferred by AutoML.

[4]:
```
X.ww.init(semantic_tags={"job": "category"}, logical_types={"job": "Categorical"})
y = ww.init_series(y)
X.ww
```

[4]:
```
       Physical Type     Logical Type Semantic Tag(s)
Column
job         category      Categorical    ['category']
state       category      Categorical    ['category']
zip            Int64  IntegerNullable     ['numeric']
action      category      Categorical    ['category']
amount       float64           Double     ['numeric']
```

### 3.2.3 Search for the best pipeline

In order to validate the results of the pipeline creation and optimization process, we will save some of our data as a holdout set.

EvalML natively supports one-hot encoding and imputation so the above `NaN` and categorical values will be taken care of.

[5]:
```
X_train, X_holdout, y_train, y_holdout = evalml.preprocessing.split_data(
    X, y, problem_type="binary", test_size=0.2, random_seed=0
)

X.ww
```

[5]:
```
       Physical Type     Logical Type Semantic Tag(s)
Column
job         category      Categorical    ['category']
state       category      Categorical    ['category']
zip            Int64  IntegerNullable     ['numeric']
action      category      Categorical    ['category']
amount       float64           Double     ['numeric']
```

Because the lead scoring labels are binary, we will use set the problem type to "binary". When we call `.search()`, the search for the best pipeline will begin.

[6]:
```
automl = AutoMLSearch(
    X_train=X_train,
    y_train=y_train,
    problem_type="binary",
    objective=lead_scoring_objective,
    additional_objectives=["auc"],
    allowed_model_families=["catboost", "random_forest", "linear_model"],
    max_batches=3,
    verbose=True,
)

automl.search(interactive_plot=False)
```

```
AutoMLSearch will use mean CV score to rank pipelines.

*****************************
* Beginning pipeline search *
*****************************

Optimizing for Lead Scoring.
Greater score is better.

Using SequentialEngine to train and score pipelines.
Searching up to 3 batches for a total of None pipelines.
Allowed model families:

Evaluating Baseline Pipeline: Mode Baseline Binary Classification Pipeline
Mode Baseline Binary Classification Pipeline:
        Starting cross validation
        Finished cross validation - mean Lead Scoring: 0.000
```

(continues on next page)

(continued from previous page)

```
*****************************
* Evaluating Batch Number 1 *
*****************************

Random Forest Classifier w/ Label Encoder + Imputer + One Hot Encoder + Oversampler:
        Starting cross validation
        Finished cross validation - mean Lead Scoring: 0.023

*****************************
* Evaluating Batch Number 2 *
*****************************

Random Forest Classifier w/ Label Encoder + Imputer + One Hot Encoder + Oversampler + RF␣
↪Classifier Select From Model:
        Starting cross validation
        Finished cross validation - mean Lead Scoring: 0.017

*****************************
* Evaluating Batch Number 3 *
*****************************

Decision Tree Classifier w/ Label Encoder + Select Columns By Type Transformer + Label␣
↪Encoder + Imputer + Select Columns Transformer + Select Columns Transformer + Label␣
↪Encoder + Imputer + One Hot Encoder + Oversampler:
        Starting cross validation
        Finished cross validation - mean Lead Scoring: 0.043
LightGBM Classifier w/ Label Encoder + Select Columns By Type Transformer + Label␣
↪Encoder + Imputer + Select Columns Transformer + Select Columns Transformer + Label␣
↪Encoder + Imputer + One Hot Encoder + Oversampler:
        Starting cross validation
        Finished cross validation - mean Lead Scoring: 0.000
Extra Trees Classifier w/ Label Encoder + Select Columns By Type Transformer + Label␣
↪Encoder + Imputer + Select Columns Transformer + Select Columns Transformer + Label␣
↪Encoder + Imputer + One Hot Encoder + Oversampler:
        Starting cross validation
        Finished cross validation - mean Lead Scoring: 0.018
Elastic Net Classifier w/ Label Encoder + Select Columns By Type Transformer + Label␣
↪Encoder + Imputer + Standard Scaler + Select Columns Transformer + Select Columns␣
↪Transformer + Label Encoder + Imputer + One Hot Encoder + Standard Scaler +␣
↪Oversampler:
        Starting cross validation
        Finished cross validation - mean Lead Scoring: 0.012
CatBoost Classifier w/ Label Encoder + Select Columns By Type Transformer + Label␣
↪Encoder + Imputer + Select Columns Transformer + Select Columns Transformer + Label␣
↪Encoder + Imputer + Oversampler:
        Starting cross validation
        Finished cross validation - mean Lead Scoring: 0.505
XGBoost Classifier w/ Label Encoder + Select Columns By Type Transformer + Label Encoder␣
↪+ Imputer + Select Columns Transformer + Select Columns Transformer + Label Encoder +␣
↪Imputer + One Hot Encoder + Oversampler:
        Starting cross validation
```

(continues on next page)

(continued from previous page)

```
        Finished cross validation - mean Lead Scoring: -0.003
Logistic Regression Classifier w/ Label Encoder + Select Columns By Type Transformer +␣
↪Label Encoder + Imputer + Standard Scaler + Select Columns Transformer + Select␣
↪Columns Transformer + Label Encoder + Imputer + One Hot Encoder + Standard Scaler +␣
↪Oversampler:
        Starting cross validation
        Finished cross validation - mean Lead Scoring: 0.019

Search finished after 00:29
Best pipeline: CatBoost Classifier w/ Label Encoder + Select Columns By Type Transformer␣
↪+ Label Encoder + Imputer + Select Columns Transformer + Select Columns Transformer +␣
↪Label Encoder + Imputer + Oversampler
Best pipeline Lead Scoring: 0.505483
```

Data type cannot be displayed: application/vnd.plotly.v1+json, text/html

[6]:
```
{1: {'Random Forest Classifier w/ Label Encoder + Imputer + One Hot Encoder + Oversampler
↪': '00:03',
  'Total time of batch': '00:03'},
 2: {'Random Forest Classifier w/ Label Encoder + Imputer + One Hot Encoder +␣
↪Oversampler + RF Classifier Select From Model': '00:03',
  'Total time of batch': '00:03'},
 3: {'Decision Tree Classifier w/ Label Encoder + Select Columns By Type Transformer +␣
↪Label Encoder + Imputer + Select Columns Transformer + Select Columns Transformer +␣
↪Label Encoder + Imputer + One Hot Encoder + Oversampler': '00:02',
  'LightGBM Classifier w/ Label Encoder + Select Columns By Type Transformer + Label␣
↪Encoder + Imputer + Select Columns Transformer + Select Columns Transformer + Label␣
↪Encoder + Imputer + One Hot Encoder + Oversampler': '00:02',
  'Extra Trees Classifier w/ Label Encoder + Select Columns By Type Transformer + Label␣
↪Encoder + Imputer + Select Columns Transformer + Select Columns Transformer + Label␣
↪Encoder + Imputer + One Hot Encoder + Oversampler': '00:03',
  'Elastic Net Classifier w/ Label Encoder + Select Columns By Type Transformer + Label␣
↪Encoder + Imputer + Standard Scaler + Select Columns Transformer + Select Columns␣
↪Transformer + Label Encoder + Imputer + One Hot Encoder + Standard Scaler + Oversampler
↪': '00:03',
  'CatBoost Classifier w/ Label Encoder + Select Columns By Type Transformer + Label␣
↪Encoder + Imputer + Select Columns Transformer + Select Columns Transformer + Label␣
↪Encoder + Imputer + Oversampler': '00:01',
  'XGBoost Classifier w/ Label Encoder + Select Columns By Type Transformer + Label␣
↪Encoder + Imputer + Select Columns Transformer + Select Columns Transformer + Label␣
↪Encoder + Imputer + One Hot Encoder + Oversampler': '00:03',
  'Logistic Regression Classifier w/ Label Encoder + Select Columns By Type Transformer␣
↪+ Label Encoder + Imputer + Standard Scaler + Select Columns Transformer + Select␣
↪Columns Transformer + Label Encoder + Imputer + One Hot Encoder + Standard Scaler +␣
↪Oversampler': '00:04',
  'Total time of batch': '00:21'}}
```

### View rankings and select pipeline

Once the fitting process is done, we can see all of the pipelines that were searched, ranked by their score on the lead scoring objective we defined.

[7]:
```
automl.rankings
```

[7]:
```
   id                                     pipeline_name  search_order  \
0   7  CatBoost Classifier w/ Label Encoder + Select ...             7
1   3  Decision Tree Classifier w/ Label Encoder + Se...             3
2   1  Random Forest Classifier w/ Label Encoder + Im...             1
3   9  Logistic Regression Classifier w/ Label Encode...             9
4   5  Extra Trees Classifier w/ Label Encoder + Sele...             5
5   2  Random Forest Classifier w/ Label Encoder + Im...             2
6   6  Elastic Net Classifier w/ Label Encoder + Sele...             6
7   0       Mode Baseline Binary Classification Pipeline             0
8   4  LightGBM Classifier w/ Label Encoder + Select ...             4
9   8  XGBoost Classifier w/ Label Encoder + Select C...             8

   ranking_score  mean_cv_score  standard_deviation_cv_score  \
0       0.505483       0.505483                     0.015122
1       0.043020       0.043020                     0.041978
2       0.022581       0.022581                     0.039111
3       0.019360       0.019360                     0.032740
4       0.018280       0.018280                     0.031661
5       0.017204       0.017204                     0.029799
6       0.011828       0.011828                     0.035386
7       0.000000       0.000000                     0.000000
8       0.000000       0.000000                     0.000000
9      -0.003226      -0.003226                     0.005587

   percent_better_than_baseline  high_variance_cv  \
0                           inf             False
1                           inf             False
2                           inf             False
3                           inf             False
4                           inf             False
5                           inf             False
6                           inf             False
7                           0.0             False
8                           0.0             False
9                           inf             False

                                        parameters
0  {'Label Encoder': {'positive_label': None}, 'N...
1  {'Label Encoder': {'positive_label': None}, 'N...
2  {'Label Encoder': {'positive_label': None}, 'I...
3  {'Label Encoder': {'positive_label': None}, 'N...
4  {'Label Encoder': {'positive_label': None}, 'N...
5  {'Label Encoder': {'positive_label': None}, 'I...
6  {'Label Encoder': {'positive_label': None}, 'N...
7  {'Label Encoder': {'positive_label': None}, 'B...
8  {'Label Encoder': {'positive_label': None}, 'N...
9  {'Label Encoder': {'positive_label': None}, 'N...
```

To select the best pipeline we can call `automl.best_pipeline`.

[8]:
```
best_pipeline = automl.best_pipeline
```

### Describe pipeline

You can get more details about any pipeline, including how it performed on other objective functions by calling `.describe_pipeline()` and specifying the `id` of the pipeline.

[9]:
```
automl.describe_pipeline(automl.rankings.iloc[0]["id"])
```

```
*****************************************************************************************************
* CatBoost Classifier w/ Label Encoder + Select Columns By Type Transformer + Label␣
↪Encoder + Imputer + Select Columns Transformer + Select Columns Transformer + Label␣
↪Encoder + Imputer + Oversampler *
*****************************************************************************************************

Problem Type: binary
Model Family: CatBoost

Pipeline Steps
==============
1. Label Encoder
         * positive_label : None
2. Select Columns By Type Transformer
         * column_types : ['category', 'EmailAddress', 'URL']
         * exclude : True
3. Label Encoder
         * positive_label : None
4. Imputer
         * categorical_impute_strategy : most_frequent
         * numeric_impute_strategy : mean
         * boolean_impute_strategy : most_frequent
         * categorical_fill_value : None
         * numeric_fill_value : None
         * boolean_fill_value : None
5. Select Columns Transformer
         * columns : ['zip', 'amount']
6. Select Columns Transformer
         * columns : ['job', 'state', 'action']
7. Label Encoder
         * positive_label : None
8. Imputer
         * categorical_impute_strategy : most_frequent
         * numeric_impute_strategy : mean
         * boolean_impute_strategy : most_frequent
         * categorical_fill_value : None
         * numeric_fill_value : None
         * boolean_fill_value : None
9. Oversampler
         * sampling_ratio : 0.25
         * k_neighbors_default : 5
```

(continues on next page)

(continued from previous page)

```
        * n_jobs : -1
        * sampling_ratio_dict : None
        * categorical_features : [2, 3, 4]
        * k_neighbors : 5
10. CatBoost Classifier
        * n_estimators : 10
        * eta : 0.03
        * max_depth : 6
        * bootstrap_type : None
        * silent : True
        * allow_writing_files : False
        * n_jobs : -1

Training
========
Training for binary problems.
Objective to optimize binary classification pipeline thresholds for: <evalml.objectives.
↪lead_scoring.LeadScoring object at 0x7f6120a75670>
Total training time (including CV): 1.3 seconds

Cross Validation
----------------
             Lead Scoring   AUC # Training # Validation
0                   0.523 0.879      3,099        1,550
1                   0.500 0.868      3,099        1,550
2                   0.494 0.897      3,100        1,549
mean                0.505 0.881          -            -
std                 0.015 0.014          -            -
coef of var         0.030 0.016          -            -
```

### 3.2.4 Evaluate on hold out

Finally, since the best pipeline was trained on all of the training data, we evaluate it on the holdout dataset.

[10]:
```
best_pipeline_score = best_pipeline.score(
    X_holdout, y_holdout, objectives=["auc", lead_scoring_objective]
)
best_pipeline_score
```

[10]:
```
OrderedDict([('AUC', 0.873904502870958),
             ('Lead Scoring', 0.37403267411865865)])
```

### 3.2.5 Why optimize for a problem-specific objective?

To demonstrate the importance of optimizing for the right objective, let's search for another pipeline using AUC, a common machine learning metric. After that, we will score the holdout data using the lead scoring objective to see how the best pipelines compare.

[11]:
```
automl_auc = evalml.AutoMLSearch(
    X_train=X_train,
    y_train=y_train,
    problem_type="binary",
    objective="auc",
    additional_objectives=[lead_scoring_objective],
    allowed_model_families=["catboost", "random_forest", "linear_model"],
    max_batches=3,
    verbose=True,
)

automl_auc.search(interactive_plot=False)
```

```
AutoMLSearch will use mean CV score to rank pipelines.

*****************************
* Beginning pipeline search *
*****************************

Optimizing for AUC.
Greater score is better.

Using SequentialEngine to train and score pipelines.
Searching up to 3 batches for a total of None pipelines.
Allowed model families:

Evaluating Baseline Pipeline: Mode Baseline Binary Classification Pipeline
Mode Baseline Binary Classification Pipeline:
        Starting cross validation
        Finished cross validation - mean AUC: 0.500

*****************************
* Evaluating Batch Number 1 *
*****************************

Random Forest Classifier w/ Label Encoder + Imputer + One Hot Encoder + Oversampler:
        Starting cross validation
        Finished cross validation - mean AUC: 0.652

*****************************
* Evaluating Batch Number 2 *
*****************************

Random Forest Classifier w/ Label Encoder + Imputer + One Hot Encoder + Oversampler + RF␣
↪Classifier Select From Model:
        Starting cross validation
        Finished cross validation - mean AUC: 0.646
```

(continues on next page)

(continued from previous page)

```
*****************************
* Evaluating Batch Number 3 *
*****************************

Decision Tree Classifier w/ Label Encoder + Select Columns By Type Transformer + Label␣
↪Encoder + Imputer + Select Columns Transformer + Select Columns Transformer + Label␣
↪Encoder + Imputer + One Hot Encoder + Oversampler:
        Starting cross validation
        Finished cross validation - mean AUC: 0.629
LightGBM Classifier w/ Label Encoder + Select Columns By Type Transformer + Label␣
↪Encoder + Imputer + Select Columns Transformer + Select Columns Transformer + Label␣
↪Encoder + Imputer + One Hot Encoder + Oversampler:
        Starting cross validation
        Finished cross validation - mean AUC: 0.625
Extra Trees Classifier w/ Label Encoder + Select Columns By Type Transformer + Label␣
↪Encoder + Imputer + Select Columns Transformer + Select Columns Transformer + Label␣
↪Encoder + Imputer + One Hot Encoder + Oversampler:
        Starting cross validation
        Finished cross validation - mean AUC: 0.653
Elastic Net Classifier w/ Label Encoder + Select Columns By Type Transformer + Label␣
↪Encoder + Imputer + Standard Scaler + Select Columns Transformer + Select Columns␣
↪Transformer + Label Encoder + Imputer + One Hot Encoder + Standard Scaler +␣
↪Oversampler:
        Starting cross validation
        Finished cross validation - mean AUC: 0.646
CatBoost Classifier w/ Label Encoder + Select Columns By Type Transformer + Label␣
↪Encoder + Imputer + Select Columns Transformer + Select Columns Transformer + Label␣
↪Encoder + Imputer + Oversampler:
        Starting cross validation
        Finished cross validation - mean AUC: 0.881
XGBoost Classifier w/ Label Encoder + Select Columns By Type Transformer + Label Encoder␣
↪+ Imputer + Select Columns Transformer + Select Columns Transformer + Label Encoder +␣
↪Imputer + One Hot Encoder + Oversampler:
        Starting cross validation
        Finished cross validation - mean AUC: 0.641
Logistic Regression Classifier w/ Label Encoder + Select Columns By Type Transformer +␣
↪Label Encoder + Imputer + Standard Scaler + Select Columns Transformer + Select␣
↪Columns Transformer + Label Encoder + Imputer + One Hot Encoder + Standard Scaler +␣
↪Oversampler:
        Starting cross validation
        Finished cross validation - mean AUC: 0.647

Search finished after 00:34
Best pipeline: CatBoost Classifier w/ Label Encoder + Select Columns By Type Transformer␣
↪+ Label Encoder + Imputer + Select Columns Transformer + Select Columns Transformer +␣
↪Label Encoder + Imputer + Oversampler
Best pipeline AUC: 0.881234
```

Data type cannot be displayed: application/vnd.plotly.v1+json, text/html

[11]:
```
{1: {'Random Forest Classifier w/ Label Encoder + Imputer + One Hot Encoder + Oversampler
↪': '00:04',
  'Total time of batch': '00:04'},
 2: {'Random Forest Classifier w/ Label Encoder + Imputer + One Hot Encoder +␣
↪Oversampler + RF Classifier Select From Model': '00:04',
  'Total time of batch': '00:04'},
 3: {'Decision Tree Classifier w/ Label Encoder + Select Columns By Type Transformer +␣
↪Label Encoder + Imputer + Select Columns Transformer + Select Columns Transformer +␣
↪Label Encoder + Imputer + One Hot Encoder + Oversampler': '00:02',
  'LightGBM Classifier w/ Label Encoder + Select Columns By Type Transformer + Label␣
↪Encoder + Imputer + Select Columns Transformer + Select Columns Transformer + Label␣
↪Encoder + Imputer + One Hot Encoder + Oversampler': '00:03',
  'Extra Trees Classifier w/ Label Encoder + Select Columns By Type Transformer + Label␣
↪Encoder + Imputer + Select Columns Transformer + Select Columns Transformer + Label␣
↪Encoder + Imputer + One Hot Encoder + Oversampler': '00:04',
  'Elastic Net Classifier w/ Label Encoder + Select Columns By Type Transformer + Label␣
↪Encoder + Imputer + Standard Scaler + Select Columns Transformer + Select Columns␣
↪Transformer + Label Encoder + Imputer + One Hot Encoder + Standard Scaler + Oversampler
↪': '00:03',
  'CatBoost Classifier w/ Label Encoder + Select Columns By Type Transformer + Label␣
↪Encoder + Imputer + Select Columns Transformer + Select Columns Transformer + Label␣
↪Encoder + Imputer + Oversampler': '00:01',
  'XGBoost Classifier w/ Label Encoder + Select Columns By Type Transformer + Label␣
↪Encoder + Imputer + Select Columns Transformer + Select Columns Transformer + Label␣
↪Encoder + Imputer + One Hot Encoder + Oversampler': '00:04',
  'Logistic Regression Classifier w/ Label Encoder + Select Columns By Type Transformer␣
↪+ Label Encoder + Imputer + Standard Scaler + Select Columns Transformer + Select␣
↪Columns Transformer + Label Encoder + Imputer + One Hot Encoder + Standard Scaler +␣
↪Oversampler': '00:03',
  'Total time of batch': '00:25'}}
```

[12]:
```
automl_auc.rankings
```

[12]:
```
   id                                     pipeline_name  search_order  \
0   7  CatBoost Classifier w/ Label Encoder + Select ...             7
1   5  Extra Trees Classifier w/ Label Encoder + Sele...             5
2   1  Random Forest Classifier w/ Label Encoder + Im...             1
3   9  Logistic Regression Classifier w/ Label Encode...             9
4   2  Random Forest Classifier w/ Label Encoder + Im...             2
5   6  Elastic Net Classifier w/ Label Encoder + Sele...             6
6   8  XGBoost Classifier w/ Label Encoder + Select C...             8
7   3  Decision Tree Classifier w/ Label Encoder + Se...             3
8   4  LightGBM Classifier w/ Label Encoder + Select ...             4
9   0       Mode Baseline Binary Classification Pipeline             0

   ranking_score  mean_cv_score  standard_deviation_cv_score  \
0       0.881234       0.881234                     0.014260
1       0.653133       0.653133                     0.058096
2       0.652452       0.652452                     0.061740
3       0.646769       0.646769                     0.043803
4       0.645598       0.645598                     0.053493
5       0.645526       0.645526                     0.042658
6       0.640950       0.640950                     0.034003
```

(continues on next page)

(continued from previous page)

```
7       0.629346       0.629346                    0.030321
8       0.624908       0.624908                    0.029747
9       0.500000       0.500000                    0.000000

   percent_better_than_baseline  high_variance_cv  \
0                     38.123362             False
1                     15.313288             False
2                     15.245206             False
3                     14.676904             False
4                     14.559799             False
5                     14.552615             False
6                     14.095050             False
7                     12.934636             False
8                     12.490777             False
9                      0.000000             False

                                          parameters
0  {'Label Encoder': {'positive_label': None}, 'N...
1  {'Label Encoder': {'positive_label': None}, 'N...
2  {'Label Encoder': {'positive_label': None}, 'I...
3  {'Label Encoder': {'positive_label': None}, 'N...
4  {'Label Encoder': {'positive_label': None}, 'I...
5  {'Label Encoder': {'positive_label': None}, 'N...
6  {'Label Encoder': {'positive_label': None}, 'N...
7  {'Label Encoder': {'positive_label': None}, 'N...
8  {'Label Encoder': {'positive_label': None}, 'N...
9  {'Label Encoder': {'positive_label': None}, 'B...
```

Like before, we can look at the rankings and pick the best pipeline.

[13]:
```
best_pipeline_auc = automl_auc.best_pipeline
```

[14]:
```
# get the auc and lead scoring score on holdout data
best_pipeline_auc_score = best_pipeline_auc.score(
    X_holdout, y_holdout, objectives=["auc", lead_scoring_objective]
)
best_pipeline_auc_score
```

[14]:
```
OrderedDict([('AUC', 0.873904502870958),
             ('Lead Scoring', 0.37403267411865865)])
```

[15]:
```
assert best_pipeline_score["Lead Scoring"] >= best_pipeline_auc_score["Lead Scoring"]
assert best_pipeline_auc_score["Lead Scoring"] >= 0
```

When we optimize for AUC, we can see that the AUC score from this pipeline is similar to the AUC score from the pipeline optimized for lead scoring. However, the revenue per lead is much smaller per lead when optimized for AUC and was much larger when optimized for lead scoring. As a result, we would have a huge gain on the amount of revenue if we optimized for lead scoring.

This happens because optimizing for AUC does not take into account the user-specified `true_positive` (dollar amount to be gained with a successful lead) and `false_positive` (dollar amount to be lost with an unsuccessful lead) values. Thus, the best pipelines may produce the highest AUC but may not actually generate the most revenue through lead scoring.

This example highlights how performance in the real world can diverge greatly from machine learning metrics.

## 3.3 Using the Cost-Benefit Matrix Objective

The Cost-Benefit Matrix (`CostBenefitMatrix`) objective is an objective that assigns costs to each of the quadrants of a confusion matrix to quantify the cost of being correct or incorrect.

### 3.3.1 Confusion Matrix

Confusion matrices are tables that summarize the number of correct and incorrectly-classified predictions, broken down by each class. They allow us to quickly understand the performance of a classification model and where the model gets "confused" when it is making predictions. For the binary classification problem, there are four possible combinations of prediction and actual target values possible:

- true positives (correct positive assignments)
- true negatives (correct negative assignments)
- false positives (incorrect positive assignments)
- false negatives (incorrect negative assignments)

An example of how to calculate a confusion matrix can be found *here*.

### 3.3.2 Cost-Benefit Matrix

Although the confusion matrix is an incredibly useful visual for understanding our model, each prediction that is correctly or incorrectly classified is treated equally. For example, for detecting breast cancer, the confusion matrix does not take into consideration that it could be much more costly to incorrectly classify a malignant tumor as benign than it is to incorrectly classify a benign tumor as malignant. This is where the cost-benefit matrix shines: it uses the cost of each of the four possible outcomes to weigh each outcome differently. By scoring using the cost-benefit matrix, we can measure the score of the model by a concrete unit that is more closely related to the goal of the model. In the below example, we will show how the cost-benefit matrix objective can be used, and how it can give us better real-world impact when compared to using other standard machine learning objectives.

### 3.3.3 Customer Churn Example

#### Data

In this example, we will be using a customer churn data set taken from Kaggle.

This dataset includes records of over 7000 customers, and includes customer account information, demographic information, services they signed up for, and whether or not the customer "churned" or left within the last month.

The target we want to predict is whether the customer churned ("Yes") or did not churn ("No"). In the dataset, approximately 73.5% of customers did not churn, and 26.5% did. We will refer to the customers who churned as the "positive" class and the customers who did not churn as the "negative" class.

[1]:
```
from evalml.demos.churn import load_churn
from evalml.preprocessing import split_data

X, y = load_churn()
```

(continues on next page)

(continued from previous page)

```
X.ww.set_types(
    {"PaymentMethod": "Categorical", "Contract": "Categorical"}
)  # Update data types Woodwork did not correctly infer
X_train, X_holdout, y_train, y_holdout = split_data(
    X, y, problem_type="binary", test_size=0.3, random_seed=0
)
```

```
             Number of Features
Categorical                  16
Numeric                       3

Number of training examples: 7043
Targets
No     73.46%
Yes    26.54%
Name: Churn, dtype: object
```

In this example, let's say that correctly identifying customers who will churn (true positive case) will give us a net profit of \$400, because it allows us to intervene, incentivize the customer to stay, and sign a new contract. Incorrectly classifying customers who were not going to churn as customers who will churn (false positive case) will cost \$100 to represent the marketing and effort used to try to retain the user. Not identifying customers who will churn (false negative case) will cost us \$200 to represent the lost in revenue from losing a customer. Finally, correctly identifying customers who will not churn (true negative case) will not cost us anything (\$0), as nothing needs to be done for that customer.

We can represent these values in our `CostBenefitMatrix` objective, where a negative value represents a cost and a positive value represents a profit–note that this means that the greater the score, the more profit we will make.

[2]:
```
from evalml.objectives import CostBenefitMatrix

cost_benefit_matrix = CostBenefitMatrix(
    true_positive=400, true_negative=0, false_positive=-100, false_negative=-200
)
```

### AutoML Search with Log Loss

First, let us run AutoML search to train pipelines using the default objective for binary classification (log loss).

[3]:
```
from evalml import AutoMLSearch

automl = AutoMLSearch(
    X_train=X_train,
    y_train=y_train,
    problem_type="binary",
    objective="log loss binary",
    max_iterations=5,
    verbose=True,
)
automl.search(interactive_plot=False)

ll_pipeline = automl.best_pipeline
ll_pipeline.score(X_holdout, y_holdout, ["log loss binary"])
```

```
AutoMLSearch will use mean CV score to rank pipelines.

*****************************
* Beginning pipeline search *
*****************************

Optimizing for Log Loss Binary.
Lower score is better.

Using SequentialEngine to train and score pipelines.
Searching up to 5 pipelines.
Allowed model families:

Evaluating Baseline Pipeline: Mode Baseline Binary Classification Pipeline
Mode Baseline Binary Classification Pipeline:
        Starting cross validation
        Finished cross validation - mean Log Loss Binary: 9.563

*****************************
* Evaluating Batch Number 1 *
*****************************

Random Forest Classifier w/ Label Encoder + Imputer + One Hot Encoder:
        Starting cross validation
        Finished cross validation - mean Log Loss Binary: 0.422

*****************************
* Evaluating Batch Number 2 *
*****************************

Random Forest Classifier w/ Label Encoder + Imputer + One Hot Encoder + RF Classifier␣
↪Select From Model:
        Starting cross validation
        Finished cross validation - mean Log Loss Binary: 0.424

*****************************
* Evaluating Batch Number 3 *
*****************************

Decision Tree Classifier w/ Label Encoder + Select Columns By Type Transformer + Label␣
↪Encoder + Imputer + Select Columns Transformer + Select Columns Transformer + Label␣
↪Encoder + Imputer + One Hot Encoder:
        Starting cross validation
        Finished cross validation - mean Log Loss Binary: 0.779
LightGBM Classifier w/ Label Encoder + Select Columns By Type Transformer + Label␣
↪Encoder + Imputer + Select Columns Transformer + Select Columns Transformer + Label␣
↪Encoder + Imputer + One Hot Encoder:
        Starting cross validation
        Finished cross validation - mean Log Loss Binary: 0.472

Search finished after 00:11
Best pipeline: Random Forest Classifier w/ Label Encoder + Imputer + One Hot Encoder
Best pipeline Log Loss Binary: 0.421573
```

Data type cannot be displayed: application/vnd.plotly.v1+json, text/html

[3]:
```
OrderedDict([('Log Loss Binary', 0.4186492571118683)])
```

When we train our pipelines using log loss as our primary objective, we try to find pipelines that minimize log loss. However, our ultimate goal in training models is to find a model that gives us the most profit, so let's score our pipeline on the cost benefit matrix (using the costs outlined above) to determine the profit we would earn from the predictions made by this model:

[4]:
```
ll_pipeline_score = ll_pipeline.score(X_holdout, y_holdout, [cost_benefit_matrix])
print(ll_pipeline_score)
```

```
OrderedDict([('Cost Benefit Matrix', 48.60388073828679)])
```

[5]:
```
# Calculate total profit across all customers using pipeline optimized for Log Loss
total_profit_ll = ll_pipeline_score["Cost Benefit Matrix"] * len(X)
print(total_profit_ll)
```

```
342317.13203975384
```

### AutoML Search with Cost-Benefit Matrix

Let's try rerunning our AutoML search, but this time using the cost-benefit matrix as our primary objective to optimize.

[6]:
```
automl = AutoMLSearch(
    X_train=X_train,
    y_train=y_train,
    problem_type="binary",
    objective=cost_benefit_matrix,
    max_iterations=5,
    verbose=True,
)
automl.search(interactive_plot=False)

cbm_pipeline = automl.best_pipeline
```

```
AutoMLSearch will use mean CV score to rank pipelines.

*****************************
* Beginning pipeline search *
*****************************

Optimizing for Cost Benefit Matrix.
Greater score is better.

Using SequentialEngine to train and score pipelines.
Searching up to 5 pipelines.
Allowed model families:

Evaluating Baseline Pipeline: Mode Baseline Binary Classification Pipeline
Mode Baseline Binary Classification Pipeline:
```

(continues on next page)

(continued from previous page)

```
        Starting cross validation
        Finished cross validation - mean Cost Benefit Matrix: -53.063

*****************************
* Evaluating Batch Number 1 *
*****************************

Random Forest Classifier w/ Label Encoder + Imputer + One Hot Encoder:
        Starting cross validation
        Finished cross validation - mean Cost Benefit Matrix: 59.575

*****************************
* Evaluating Batch Number 2 *
*****************************

Random Forest Classifier w/ Label Encoder + Imputer + One Hot Encoder + RF Classifier␣
↪Select From Model:
        Starting cross validation
        Finished cross validation - mean Cost Benefit Matrix: 56.796

*****************************
* Evaluating Batch Number 3 *
*****************************

Decision Tree Classifier w/ Label Encoder + Select Columns By Type Transformer + Label␣
↪Encoder + Imputer + Select Columns Transformer + Select Columns Transformer + Label␣
↪Encoder + Imputer + One Hot Encoder:
        Starting cross validation
        Finished cross validation - mean Cost Benefit Matrix: 55.903
LightGBM Classifier w/ Label Encoder + Select Columns By Type Transformer + Label␣
↪Encoder + Imputer + Select Columns Transformer + Select Columns Transformer + Label␣
↪Encoder + Imputer + One Hot Encoder:
        Starting cross validation
        Finished cross validation - mean Cost Benefit Matrix: 52.942

Search finished after 00:17
Best pipeline: Random Forest Classifier w/ Label Encoder + Imputer + One Hot Encoder
Best pipeline Cost Benefit Matrix: 59.574683
```

Data type cannot be displayed: application/vnd.plotly.v1+json, text/html

Now, if we calculate the cost-benefit matrix score on our best pipeline, we see that with this pipeline optimized for our cost-benefit matrix objective, we are able to generate more profit per customer. Across our 7043 customers, we generate much more profit using this best pipeline! Custom objectives like `CostBenefitMatrix` are just one example of how using EvalML can help find pipelines that can perform better on real-world problems, rather than on arbitrary standard statistical metrics.

[7]:
```
cbm_pipeline_score = cbm_pipeline.score(X_holdout, y_holdout, [cost_benefit_matrix])
print(cbm_pipeline_score)
```

```
OrderedDict([('Cost Benefit Matrix', 57.40653099858022)])
```

[8]:
```
# Calculate total profit across all customers using pipeline optimized for␣
↪CostBenefitMatrix
total_profit_cbm = cbm_pipeline_score["Cost Benefit Matrix"] * len(X)
print(total_profit_cbm)
```

```
404314.1978230005
```

[9]:
```
# Calculate difference in profit made using both pipelines
profit_diff = total_profit_cbm - total_profit_ll
print(profit_diff)
```

```
61997.06578324665
```

Finally, we can graph the confusion matrices for both pipelines to better understand why the pipeline trained using the cost-benefit matrix is able to correctly classify more samples than the pipeline trained with log loss: we were able to correctly predict more cases where the customer would have churned (true positive), allowing us to intervene and prevent those customers from leaving.

[10]:
```
from evalml.model_understanding.metrics import graph_confusion_matrix

# pipeline trained with log loss
y_pred = ll_pipeline.predict(X_holdout)
graph_confusion_matrix(y_holdout, y_pred)
```

Data type cannot be displayed: application/vnd.plotly.v1+json, text/html

[11]:
```
# pipeline trained with cost-benefit matrix
y_pred = cbm_pipeline.predict(X_holdout)
graph_confusion_matrix(y_holdout, y_pred)
```

Data type cannot be displayed: application/vnd.plotly.v1+json, text/html

## 3.4 Using Text Data with EvalML

In this demo, we will show you how to use EvalML to build models which use text data.

[1]:
```
import evalml
from evalml import AutoMLSearch
```

### 3.4.1 Dataset

We will be utilizing a dataset of SMS text messages, some of which are categorized as spam, and others which are not ("ham"). This dataset is originally from Kaggle, but modified to produce a slightly more even distribution of spam to ham.

[2]:
```
from urllib.request import urlopen
import pandas as pd

input_data = urlopen(
    "https://featurelabs-static.s3.amazonaws.com/spam_text_messages_modified.csv"
)
data = pd.read_csv(input_data)[:750]

X = data.drop(["Category"], axis=1)
y = data["Category"]

display(X.head())
```

```
                                            Message
0  Free entry in 2 a wkly comp to win FA Cup fina...
1  FreeMsg Hey there darling it's been 3 week's n...
2  WINNER!! As a valued network customer you have...
3  Had your mobile 11 months or more? U R entitle...
4  SIX chances to win CASH! From 100 to 20,000 po...
```

The ham vs spam distribution of the data is 3:1, so any machine learning model must get above 75% accuracy in order to perform better than a trivial baseline model which simply classifies everything as ham.

[3]:
```
y.value_counts(normalize=True)
```

[3]:
```
spam    0.593333
ham     0.406667
Name: Category, dtype: float64
```

In order to properly utilize Woodwork's 'Natural Language' typing, we need to pass this argument in during initialization. Otherwise, this will be treated as an 'Unknown' type and dropped in the search.

[4]:
```
X.ww.init(logical_types={"Message": "NaturalLanguage"})
```

### 3.4.2 Search for best pipeline

In order to validate the results of the pipeline creation and optimization process, we will save some of our data as a holdout set.

[5]:
```
X_train, X_holdout, y_train, y_holdout = evalml.preprocessing.split_data(
    X, y, problem_type="binary", test_size=0.2, random_seed=0
)
```

EvalML uses Woodwork to automatically detect which columns are text columns, so you can run search normally, as you would if there was no text data. We can print out the logical type of the `Message` column and assert that it is indeed inferred as a natural language column.

[6]:
```
X_train.ww
```

[6]:
```
        Physical Type     Logical Type Semantic Tag(s)
Column
Message        string  NaturalLanguage              []
```

Because the spam/ham labels are binary, we will use `AutoMLSearch(X_train=X_train, y_train=y_train, problem_type='binary')`. When we call `.search()`, the search for the best pipeline will begin.

[7]:
```
automl = AutoMLSearch(
    X_train=X_train,
    y_train=y_train,
    problem_type="binary",
    max_batches=1,
    optimize_thresholds=True,
    verbose=True,
)

automl.search(interactive_plot=False)
```

```
AutoMLSearch will use mean CV score to rank pipelines.

*****************************
* Beginning pipeline search *
*****************************

Optimizing for Log Loss Binary.
Lower score is better.

Using SequentialEngine to train and score pipelines.
Searching up to 1 batches for a total of None pipelines.
Allowed model families:

Evaluating Baseline Pipeline: Mode Baseline Binary Classification Pipeline
Mode Baseline Binary Classification Pipeline:
        Starting cross validation
        Finished cross validation - mean Log Loss Binary: 14.658

*****************************
* Evaluating Batch Number 1 *
*****************************

Random Forest Classifier w/ Label Encoder + Natural Language Featurizer + Imputer:
        Starting cross validation
        Finished cross validation - mean Log Loss Binary: 0.212

Search finished after 00:06
Best pipeline: Random Forest Classifier w/ Label Encoder + Natural Language Featurizer +␣
↪Imputer
Best pipeline Log Loss Binary: 0.212470
```

Data type cannot be displayed: application/vnd.plotly.v1+json, text/html

[7]:
```
{1: {'Random Forest Classifier w/ Label Encoder + Natural Language Featurizer + Imputer':
↪ '00:05',
  'Total time of batch': '00:05'}}
```

### View rankings and select pipeline

Once the fitting process is done, we can see all of the pipelines that were searched.

[8]:
```
automl.rankings
```

[8]:
```
   id                                     pipeline_name  search_order  \
0   1  Random Forest Classifier w/ Label Encoder + Na...             1
1   0       Mode Baseline Binary Classification Pipeline             0

   ranking_score  mean_cv_score  standard_deviation_cv_score  \
0       0.212470       0.212470                     0.043953
1      14.657752      14.657752                     0.104049

   percent_better_than_baseline  high_variance_cv  \
0                     98.550459             False
1                      0.000000             False

                                           parameters
0  {'Label Encoder': {'positive_label': None}, 'I...
1  {'Label Encoder': {'positive_label': None}, 'B...
```

To select the best pipeline we can call `automl.best_pipeline`.

[9]:
```
best_pipeline = automl.best_pipeline
```

### Describe pipeline

You can get more details about any pipeline, including how it performed on other objective functions.

[10]:
```
automl.describe_pipeline(automl.rankings.iloc[0]["id"])
```

```
*******************************************************************************
* Random Forest Classifier w/ Label Encoder + Natural Language Featurizer + Imputer *
*******************************************************************************

Problem Type: binary
Model Family: Random Forest

Pipeline Steps
==============
1. Label Encoder
         * positive_label : None
2. Natural Language Featurizer
3. Imputer
         * categorical_impute_strategy : most_frequent
         * numeric_impute_strategy : mean
```

(continues on next page)

(continued from previous page)

```
        * boolean_impute_strategy : most_frequent
        * categorical_fill_value : None
        * numeric_fill_value : None
        * boolean_fill_value : None
4. Random Forest Classifier
        * n_estimators : 100
        * max_depth : 6
        * n_jobs : -1

Training
========
Training for binary problems.
Total training time (including CV): 5.5 seconds

Cross Validation
----------------
             Log Loss Binary  MCC Binary  Gini   AUC  Precision    F1  Balanced Accuracy␣
↪Binary  Accuracy Binary # Training # Validation
0                      0.213       0.876 0.948 0.974      0.916 0.927                  ␣
↪ 0.940            0.940        400          200
1                      0.168       0.865 0.976 0.988      0.947 0.917                  ␣
↪ 0.928            0.935        400          200
2                      0.256       0.782 0.920 0.960      0.896 0.868                  ␣
↪ 0.887            0.895        400          200
mean                   0.212       0.841 0.948 0.974      0.920 0.904                  ␣
↪ 0.918            0.923          -            -
std                    0.044       0.051 0.028 0.014      0.026 0.032                  ␣
↪ 0.028            0.025          -            -
coef of var            0.207       0.061 0.030 0.014      0.028 0.035                  ␣
↪ 0.030            0.027          -            -
```

[11]:
```
best_pipeline.graph()
```

[11]:

Notice above that there is a `Natural Language Featurizer` as the first step in the pipeline. AutoMLSearch uses the woodwork accessor to recognize that `'Message'` is a text column, and converts this text into numerical values that can be handled by the estimator.

### 3.4.3 Evaluate on holdout

Now, we can score the pipeline on the holdout data using the ranking objectives for binary classification problems.

[12]:
```
scores = best_pipeline.score(
    X_holdout, y_holdout, objectives=evalml.objectives.get_ranking_objectives("binary")
)
print(f'Accuracy Binary: {scores["Accuracy Binary"]}')
```

```
Accuracy Binary: 0.9333333333333333
```

As you can see, this model performs relatively well on this dataset, even on unseen data.

### 3.4.4 What does the Natural Language Featurizer do?

Machine learning models cannot handle non-numeric data. Any text must be broken down into numeric features that provide useful information about that text. The Natural Natural Language Featurizer first normalizes your text by removing any punctuation and other non-alphanumeric characters and converting any capital letters to lowercase. From there, it passes the text into featuretools' nlp_primitives `dfs` search, resulting in several informative features that replace the original column in your dataset: Diversity Score, Mean Characters per Word, Polarity Score, LSA (Latent Semantic Analysis), Number of Characters, and Number of Words.

**Diversity Score** is the ratio of unique words to total words.

**Mean Characters per Word** is the average number of letters in each word.

**Polarity Score** is a prediction of how "polarized" the text is, on a scale from -1 (extremely negative) to 1 (extremely positive).

**Latent Semantic Analysis** is an abstract representation of how important each word is with respect to the entire text, reduced down into two values per text. While the other text features are each a single column, this feature adds two columns to your data, `LSA(column_name)[0]` and `LSA(column_name)[1]`.

**Number of Characters** is the number of characters in the text.

**Number of Words** is the number of words in the text.

Let's see what this looks like with our spam/ham example.

[13]:
```
best_pipeline.input_feature_names
```

[13]:
```
{'Label Encoder': ['Message'],
 'Natural Language Featurizer': ['Message'],
 'Imputer': ['DIVERSITY_SCORE(Message)',
  'MEAN_CHARACTERS_PER_WORD(Message)',
  'NUM_CHARACTERS(Message)',
  'NUM_WORDS(Message)',
  'POLARITY_SCORE(Message)',
  'LSA(Message)[0]',
  'LSA(Message)[1]'],
 'Random Forest Classifier': ['DIVERSITY_SCORE(Message)',
  'MEAN_CHARACTERS_PER_WORD(Message)',
  'NUM_CHARACTERS(Message)',
  'NUM_WORDS(Message)',
  'POLARITY_SCORE(Message)',
  'LSA(Message)[0]',
  'LSA(Message)[1]']}
```

Here, the Natural Language Featurizer takes in a single "Message" column, but then the next component in the pipeline, the Imputer, receives five columns of input. These five columns are the result of featurizing the text-type "Message" column. Most importantly, these featurized columns are what ends up passed in to the estimator.

If the dataset had any non-text columns, those would be left alone by this process. If the dataset had more than one text column, each would be broken into these five feature columns independently.

### The features, more directly

Rather than just checking the new column names, let's examine the output of this component directly. We can see this by running the component on its own.

[14]:
```
natural_language_featurizer = evalml.pipelines.components.NaturalLanguageFeaturizer()
X_featurized = natural_language_featurizer.fit_transform(X_train)
```

Now we can compare the input data to the output from the Natural Language Featurizer:

[15]:
```
X_train.head()
```

[15]:
```
                                            Message
296  Sunshine Hols. To claim ur med holiday send a ...
652                                Yup ü not comin :-(
526  Hello hun how ru? Its here by the way. Im good...
571  I tagged MY friends that you seemed to count a...
472                      What happened to our yo date?
```

[16]:
```
X_featurized.head()
```

[16]:
```
     DIVERSITY_SCORE(Message)  MEAN_CHARACTERS_PER_WORD(Message)  \
296                       1.0                           4.344828
652                       1.0                           3.000000
526                       1.0                           3.363636
571                       0.8                           4.083333
472                       1.0                           3.833333

     NUM_CHARACTERS(Message)  NUM_WORDS(Message)  POLARITY_SCORE(Message)  \
296                    154.0                29.0                    0.003
652                     16.0                 4.0                    0.000
526                    143.0                33.0                    0.162
571                     60.0                12.0                    0.681
472                     28.0                 6.0                    0.000

     LSA(Message)[0]  LSA(Message)[1]
296         0.150556        -0.072443
652         0.017340        -0.005411
526         0.169954         0.022670
571         0.144713         0.036799
472         0.109373        -0.042754
```

These numeric values now represent important information about the original text that the estimator at the end of the pipeline can successfully use to make predictions.

### 3.4.5 Why encode text this way?

To demonstrate the importance of text-specific modeling, let's train a model with the same dataset, without letting `AutoMLSearch` detect the text column. We can change this by explicitly setting the data type of the `'Message'` column in Woodwork to `Categorical` using the utility method `infer_feature_types`.

[17]:
```
from evalml.utils import infer_feature_types

X = infer_feature_types(X, {"Message": "Categorical"})
X_train, X_holdout, y_train, y_holdout = evalml.preprocessing.split_data(
    X, y, problem_type="binary", test_size=0.2, random_seed=0
)
```

[18]:
```
automl_no_text = AutoMLSearch(
    X_train=X_train,
    y_train=y_train,
    problem_type="binary",
    max_batches=1,
    optimize_thresholds=True,
    verbose=True,
)

automl_no_text.search(interactive_plot=False)
```

```
AutoMLSearch will use mean CV score to rank pipelines.

*****************************
* Beginning pipeline search *
*****************************

Optimizing for Log Loss Binary.
Lower score is better.

Using SequentialEngine to train and score pipelines.
Searching up to 1 batches for a total of None pipelines.
Allowed model families:

Evaluating Baseline Pipeline: Mode Baseline Binary Classification Pipeline
Mode Baseline Binary Classification Pipeline:
        Starting cross validation
        Finished cross validation - mean Log Loss Binary: 14.658

*****************************
* Evaluating Batch Number 1 *
*****************************

Random Forest Classifier w/ Label Encoder + Natural Language Featurizer + Imputer:
        Starting cross validation
        Finished cross validation - mean Log Loss Binary: 0.212

Search finished after 00:04
Best pipeline: Random Forest Classifier w/ Label Encoder + Natural Language Featurizer +␣
↪Imputer
Best pipeline Log Loss Binary: 0.212470
```

Data type cannot be displayed: application/vnd.plotly.v1+json, text/html

[18]:
```
{1: {'Random Forest Classifier w/ Label Encoder + Natural Language Featurizer + Imputer':
↪ '00:04',
  'Total time of batch': '00:04'}}
```

Like before, we can look at the rankings and pick the best pipeline.

[19]:
```
automl_no_text.rankings
```

[19]:
```
   id                                     pipeline_name  search_order  \
0   1  Random Forest Classifier w/ Label Encoder + Na...             1
1   0       Mode Baseline Binary Classification Pipeline             0

   ranking_score  mean_cv_score  standard_deviation_cv_score  \
0       0.212470       0.212470                     0.043953
1      14.657752      14.657752                     0.104049

   percent_better_than_baseline  high_variance_cv  \
0                     98.550459             False
1                      0.000000             False

                                          parameters
0  {'Label Encoder': {'positive_label': None}, 'I...
1  {'Label Encoder': {'positive_label': None}, 'B...
```

[20]:
```
best_pipeline_no_text = automl_no_text.best_pipeline
```

Here, changing the data type of the text column removed the `Natural Language Featurizer` from the pipeline.

[21]:
```
best_pipeline_no_text.graph()
```

[21]:

[22]:
```
automl_no_text.describe_pipeline(automl_no_text.rankings.iloc[0]["id"])
```

```
*****************************************************************************************
* Random Forest Classifier w/ Label Encoder + Natural Language Featurizer + Imputer *
*****************************************************************************************

Problem Type: binary
Model Family: Random Forest

Pipeline Steps
==============
1. Label Encoder
         * positive_label : None
2. Natural Language Featurizer
3. Imputer
         * categorical_impute_strategy : most_frequent
         * numeric_impute_strategy : mean
         * boolean_impute_strategy : most_frequent
```

(continues on next page)

(continued from previous page)

```
        * categorical_fill_value : None
        * numeric_fill_value : None
        * boolean_fill_value : None
4. Random Forest Classifier
        * n_estimators : 100
        * max_depth : 6
        * n_jobs : -1

Training
========
Training for binary problems.
Total training time (including CV): 4.0 seconds

Cross Validation
----------------
             Log Loss Binary  MCC Binary  Gini   AUC  Precision    F1  Balanced Accuracy␣
↪Binary  Accuracy Binary # Training # Validation
0                      0.213       0.876 0.948 0.974      0.916 0.927                   ␣
↪ 0.940            0.940        400          200
1                      0.168       0.865 0.976 0.988      0.947 0.917                   ␣
↪ 0.928            0.935        400          200
2                      0.256       0.782 0.920 0.960      0.896 0.868                   ␣
↪ 0.887            0.895        400          200
mean                   0.212       0.841 0.948 0.974      0.920 0.904                   ␣
↪ 0.918            0.923          -            -
std                    0.044       0.051 0.028 0.014      0.026 0.032                   ␣
↪ 0.028            0.025          -            -
coef of var            0.207       0.061 0.030 0.014      0.028 0.035                   ␣
↪ 0.030            0.027          -            -
```

[23]:
```
# get standard performance metrics on holdout data
scores = best_pipeline_no_text.score(
    X_holdout, y_holdout, objectives=evalml.objectives.get_ranking_objectives("binary")
)
print(f'Accuracy Binary: {scores["Accuracy Binary"]}')
```

```
Accuracy Binary: 0.9333333333333333
```

Without the `Natural Language Featurizer`, the `'Message'` column was treated as a categorical column, and therefore the conversion of this text to numerical features happened in the `One Hot Encoder`. The best pipeline encoded the top 10 most frequent "categories" of these texts, meaning 10 text messages were one-hot encoded and all the others were dropped. Clearly, this removed almost all of the information from the dataset, as we can see the `best_pipeline_no_text` performs very similarly to randomly guessing "ham" in every case.

CHAPTER

FOUR

# USER GUIDE

These guides include in-depth descriptions and explanations of EvalML's features.

## 4.1 Automated Machine Learning (AutoML) Search

### 4.1.1 Background

#### Machine Learning

Machine learning (ML) is the process of constructing a mathematical model of a system based on a sample dataset collected from that system.

One of the main goals of training an ML model is to teach the model to separate the signal present in the data from the noise inherent in system and in the data collection process. If this is done effectively, the model can then be used to make accurate predictions about the system when presented with new, similar data. Additionally, introspecting on an ML model can reveal key information about the system being modeled, such as which inputs and transformations of the inputs are most useful to the ML model for learning the signal in the data, and are therefore the most predictive.

There are a variety of ML problem types. Supervised learning describes the case where the collected data contains an output value to be modeled and a set of inputs with which to train the model. EvalML focuses on training supervised learning models.

EvalML supports three common supervised ML problem types. The first is regression, where the target value to model is a continuous numeric value. Next are binary and multiclass classification, where the target value to model consists of two or more discrete values or categories. The choice of which supervised ML problem type is most appropriate depends on domain expertise and on how the model will be evaluated and used.

EvalML is currently building support for supervised time series problems: time series regression, time series binary classification, and time series multiclass classification. While we've added some features to tackle these kinds of problems, our functionality is still being actively developed so please be mindful of that before using it.

#### AutoML and Search

AutoML is the process of automating the construction, training and evaluation of ML models. Given a data and some configuration, AutoML searches for the most effective and accurate ML model or models to fit the dataset. During the search, AutoML will explore different combinations of model type, model parameters and model architecture.

An effective AutoML solution offers several advantages over constructing and tuning ML models by hand. AutoML can assist with many of the difficult aspects of ML, such as avoiding overfitting and underfitting, imbalanced data, detecting data leakage and other potential issues with the problem setup, and automatically applying best-practice data cleaning, feature engineering, feature selection and various modeling techniques. AutoML can also leverage search algorithms to

optimally sweep the hyperparameter search space, resulting in model performance which would be difficult to achieve by manual training.

### 4.1.2 AutoML in EvalML

EvalML supports all of the above and more.

In its simplest usage, the AutoML search interface requires only the input data, the target data and a `problem_type` specifying what kind of supervised ML problem to model.

** Graphing methods, like verbose AutoMLSearch, on Jupyter Notebook and Jupyter Lab require ipywidgets to be installed.

** If graphing on Jupyter Lab, jupyterlab-plotly required. To download this, make sure you have npm installed.

[1]:
```
import evalml
from evalml.utils import infer_feature_types

X, y = evalml.demos.load_fraud(n_rows=650)
```

```
             Number of Features
Boolean                       1
Categorical                   6
Numeric                       5

Number of training examples: 650
Targets
False    86.31%
True     13.69%
Name: fraud, dtype: object
```

To provide data to EvalML, it is recommended that you initialize a Woodwork accessor on your data. This allows you to easily control how EvalML will treat each of your features before training a model.

EvalML also accepts `pandas` input, and will run type inference on top of the input `pandas` data. If you'd like to change the types inferred by EvalML, you can use the `infer_feature_types` utility method, which takes pandas or numpy input and converts it to a Woodwork data structure. The `feature_types` parameter can be used to specify what types specific columns should be.

Feature types such as `Natural Language` must be specified in this way, otherwise Woodwork will infer it as `Unknown` type and drop it during the AutoMLSearch.

In the example below, we reformat a couple features to make them easily consumable by the model, and then specify that the provider, which would have otherwise been inferred as a column with natural language, is a categorical column.

[2]:
```
X.ww["expiration_date"] = X["expiration_date"].apply(
    lambda x: "20{}-01-{}".format(x.split("/")[1], x.split("/")[0])
)
X = infer_feature_types(
    X,
    feature_types={
        "store_id": "categorical",
        "expiration_date": "datetime",
        "lat": "categorical",
        "lng": "categorical",
        "provider": "categorical",
```

(continues on next page)

(continued from previous page)

```
    },
)
```

In order to validate the results of the pipeline creation and optimization process, we will save some of our data as a holdout set.

[3]:
```
X_train, X_holdout, y_train, y_holdout = evalml.preprocessing.split_data(
    X, y, problem_type="binary", test_size=0.2
)
```

### Data Checks

Before calling `AutoMLSearch.search`, we should run some sanity checks on our data to ensure that the input data being passed will not run into some common issues before running a potentially time-consuming search. EvalML has various data checks that makes this easy. Each data check will return a collection of warnings and errors if it detects potential issues with the input data. This allows users to inspect their data to avoid confusing errors that may arise during the search process. You can learn about each of the data checks available through our *data checks guide*.

Here, we will run the `DefaultDataChecks` class, which contains a series of data checks that are generally useful.

[4]:
```
from evalml.data_checks import DefaultDataChecks

data_checks = DefaultDataChecks("binary", "log loss binary")
data_checks.validate(X_train, y_train)
```

[4]:
```
[]
```

Since there were no warnings or errors returned, we can safely continue with the search process.

### Holdout Set for Pipeline Ranking

If the `holdout_set_size` parameter is set and the input dataset has more than 500 rows, AutoMLSearch will create a holdout set from `holdout_set_size` of the training data. Alternatively, a holdout set can be manually specified by using the `X_holdout` and `y_holdout` parameters in `AutoMLSearch()`. In this example, the holdout set created previously will be used by AutoML search.

During the AutoML search process, the mean of the objective scores of all cross validation folds (shown the "mean_cv_score" column in the pipeline rankings), is calculated. This score is passed to the AutoML search tuner to further optimize the hyperparameters of the next batch of pipelines.

After, the pipeline will be fitted on the entire training dataset and scored on this new holdout set. This score is represented under the "ranking_score" column on the pipeline rankings board and is used to rank pipeline performance.

If a dataset has less than 500 rows or `holdout_set_size=0` (which is the default setting), the "mean_cv_score" will be used as the ranking_score instead.

[5]:
```
automl = evalml.automl.AutoMLSearch(
    X_train=X_train,
    y_train=y_train,
    X_holdout=X_holdout,
    y_holdout=y_holdout,
    problem_type="binary",
    verbose=True,
```

(continues on next page)

(continued from previous page)

```
)
automl.search(interactive_plot=False)
```

```
AutoMLSearch will use the holdout set to score and rank pipelines.
Removing columns ['currency'] because they are of 'Unknown' type
Using default limit of max_batches=3.


*****************************
* Beginning pipeline search *
*****************************

Optimizing for Log Loss Binary.
Lower score is better.

Using SequentialEngine to train and score pipelines.
Searching up to 3 batches for a total of None pipelines.
Allowed model families:

Evaluating Baseline Pipeline: Mode Baseline Binary Classification Pipeline
Mode Baseline Binary Classification Pipeline:
        Starting cross validation
        Finished cross validation - mean Log Loss Binary: 4.921
        Starting holdout set scoring
        Finished holdout set scoring - Log Loss Binary: 4.991

*****************************
* Evaluating Batch Number 1 *
*****************************

Random Forest Classifier w/ Label Encoder + Drop Columns Transformer + DateTime␣
↪Featurizer + Imputer + One Hot Encoder + Oversampler:
        Starting cross validation
        Finished cross validation - mean Log Loss Binary: 0.259
        Starting holdout set scoring
        Finished holdout set scoring - Log Loss Binary: 0.224

*****************************
* Evaluating Batch Number 2 *
*****************************

Random Forest Classifier w/ Label Encoder + Drop Columns Transformer + DateTime␣
↪Featurizer + Imputer + One Hot Encoder + Oversampler + RF Classifier Select From Model:
        Starting cross validation
        Finished cross validation - mean Log Loss Binary: 0.254
        Starting holdout set scoring
        Finished holdout set scoring - Log Loss Binary: 0.219

*****************************
* Evaluating Batch Number 3 *
*****************************
```

(continues on next page)

(continued from previous page)

```
Decision Tree Classifier w/ Label Encoder + Select Columns By Type Transformer + Label␣
↪Encoder + Drop Columns Transformer + DateTime Featurizer + Imputer + Select Columns␣
↪Transformer + Select Columns Transformer + Label Encoder + Imputer + One Hot Encoder +␣
↪Oversampler:
        Starting cross validation
        Finished cross validation - mean Log Loss Binary: 1.449
        Starting holdout set scoring
        Finished holdout set scoring - Log Loss Binary: 1.003
        High coefficient of variation (cv >= 0.5) within cross validation scores.
        Decision Tree Classifier w/ Label Encoder + Select Columns By Type Transformer +␣
↪Label Encoder + Drop Columns Transformer + DateTime Featurizer + Imputer + Select␣
↪Columns Transformer + Select Columns Transformer + Label Encoder + Imputer + One Hot␣
↪Encoder + Oversampler may not perform as estimated on unseen data.
LightGBM Classifier w/ Label Encoder + Select Columns By Type Transformer + Label␣
↪Encoder + Drop Columns Transformer + DateTime Featurizer + Imputer + Select Columns␣
↪Transformer + Select Columns Transformer + Label Encoder + Imputer + One Hot Encoder +␣
↪Oversampler:
        Starting cross validation
        Finished cross validation - mean Log Loss Binary: 0.300
        Starting holdout set scoring
        Finished holdout set scoring - Log Loss Binary: 0.161
Extra Trees Classifier w/ Label Encoder + Select Columns By Type Transformer + Label␣
↪Encoder + Drop Columns Transformer + DateTime Featurizer + Imputer + Select Columns␣
↪Transformer + Select Columns Transformer + Label Encoder + Imputer + One Hot Encoder +␣
↪Oversampler:
        Starting cross validation
        Finished cross validation - mean Log Loss Binary: 0.361
        Starting holdout set scoring
        Finished holdout set scoring - Log Loss Binary: 0.348
Elastic Net Classifier w/ Label Encoder + Select Columns By Type Transformer + Label␣
↪Encoder + Drop Columns Transformer + DateTime Featurizer + Imputer + Standard Scaler +␣
↪Select Columns Transformer + Select Columns Transformer + Label Encoder + Imputer +␣
↪One Hot Encoder + Standard Scaler + Oversampler:
        Starting cross validation
        Finished cross validation - mean Log Loss Binary: 0.375
        Starting holdout set scoring
        Finished holdout set scoring - Log Loss Binary: 0.400
CatBoost Classifier w/ Label Encoder + Select Columns By Type Transformer + Label␣
↪Encoder + Drop Columns Transformer + DateTime Featurizer + Imputer + Select Columns␣
↪Transformer + Select Columns Transformer + Label Encoder + Imputer + Oversampler:
        Starting cross validation
        Finished cross validation - mean Log Loss Binary: 0.569
        Starting holdout set scoring
        Finished holdout set scoring - Log Loss Binary: 0.557
XGBoost Classifier w/ Label Encoder + Select Columns By Type Transformer + Label Encoder␣
↪+ Drop Columns Transformer + DateTime Featurizer + Imputer + Select Columns␣
↪Transformer + Select Columns Transformer + Label Encoder + Imputer + One Hot Encoder +␣
↪Oversampler:
        Starting cross validation
        Finished cross validation - mean Log Loss Binary: 0.257
        Starting holdout set scoring
        Finished holdout set scoring - Log Loss Binary: 0.142
```

(continues on next page)

(continued from previous page)

```
Logistic Regression Classifier w/ Label Encoder + Select Columns By Type Transformer +␣
↪Label Encoder + Drop Columns Transformer + DateTime Featurizer + Imputer + Standard␣
↪Scaler + Select Columns Transformer + Select Columns Transformer + Label Encoder +␣
↪Imputer + One Hot Encoder + Standard Scaler + Oversampler:
        Starting cross validation
        Finished cross validation - mean Log Loss Binary: 0.374
        Starting holdout set scoring
        Finished holdout set scoring - Log Loss Binary: 0.402

Search finished after 00:40
Best pipeline: XGBoost Classifier w/ Label Encoder + Select Columns By Type Transformer␣
↪+ Label Encoder + Drop Columns Transformer + DateTime Featurizer + Imputer + Select␣
↪Columns Transformer + Select Columns Transformer + Label Encoder + Imputer + One Hot␣
↪Encoder + Oversampler
Best pipeline Log Loss Binary: 0.142417
```

Data type cannot be displayed: application/vnd.plotly.v1+json, text/html

[5]:
```
{1: {'Random Forest Classifier w/ Label Encoder + Drop Columns Transformer + DateTime␣
↪Featurizer + Imputer + One Hot Encoder + Oversampler': '00:04',
  'Total time of batch': '00:04'},
 2: {'Random Forest Classifier w/ Label Encoder + Drop Columns Transformer + DateTime␣
↪Featurizer + Imputer + One Hot Encoder + Oversampler + RF Classifier Select From Model
↪': '00:05',
  'Total time of batch': '00:05'},
 3: {'Decision Tree Classifier w/ Label Encoder + Select Columns By Type Transformer +␣
↪Label Encoder + Drop Columns Transformer + DateTime Featurizer + Imputer + Select␣
↪Columns Transformer + Select Columns Transformer + Label Encoder + Imputer + One Hot␣
↪Encoder + Oversampler': '00:03',
  'LightGBM Classifier w/ Label Encoder + Select Columns By Type Transformer + Label␣
↪Encoder + Drop Columns Transformer + DateTime Featurizer + Imputer + Select Columns␣
↪Transformer + Select Columns Transformer + Label Encoder + Imputer + One Hot Encoder +␣
↪Oversampler': '00:03',
  'Extra Trees Classifier w/ Label Encoder + Select Columns By Type Transformer + Label␣
↪Encoder + Drop Columns Transformer + DateTime Featurizer + Imputer + Select Columns␣
↪Transformer + Select Columns Transformer + Label Encoder + Imputer + One Hot Encoder +␣
↪Oversampler': '00:04',
  'Elastic Net Classifier w/ Label Encoder + Select Columns By Type Transformer + Label␣
↪Encoder + Drop Columns Transformer + DateTime Featurizer + Imputer + Standard Scaler +␣
↪Select Columns Transformer + Select Columns Transformer + Label Encoder + Imputer +␣
↪One Hot Encoder + Standard Scaler + Oversampler': '00:04',
  'CatBoost Classifier w/ Label Encoder + Select Columns By Type Transformer + Label␣
↪Encoder + Drop Columns Transformer + DateTime Featurizer + Imputer + Select Columns␣
↪Transformer + Select Columns Transformer + Label Encoder + Imputer + Oversampler': '00:
↪02',
  'XGBoost Classifier w/ Label Encoder + Select Columns By Type Transformer + Label␣
↪Encoder + Drop Columns Transformer + DateTime Featurizer + Imputer + Select Columns␣
↪Transformer + Select Columns Transformer + Label Encoder + Imputer + One Hot Encoder +␣
↪Oversampler': '00:04',
  'Logistic Regression Classifier w/ Label Encoder + Select Columns By Type Transformer␣
↪+ Label Encoder + Drop Columns Transformer + DateTime Featurizer + Imputer + Standard␣
↪Scaler + Select Columns Transformer + Select Columns Transformer + Label Encoder +␣
↪Imputer + One Hot Encoder + Standard Scaler + Oversampler': '00:06',
```

(continues on next page)

(continued from previous page)

```
 'Total time of batch': '00:30'}}
```

With the `verbose` argument set to True, the AutoML search will log its progress, reporting each pipeline and parameter set evaluated during the search. The search iteration plot shown during AutoML search tracks the current pipeline's validation score (tracked as the gray point) against the best pipeline validation score (tracked as the blue line).

There are a number of mechanisms to control the AutoML search time. One way is to set the `max_batches` parameter which controls the maximum number of rounds of AutoML to evaluate, where each round may train and score a variable number of pipelines. Another way is to set the `max_iterations` parameter which controls the maximum number of candidate models to be evaluated during AutoML. By default, AutoML will search for a single batch. The first pipeline to be evaluated will always be a baseline model representing a trivial solution.

The AutoML interface supports a variety of other parameters. For a comprehensive list, please *refer to the API reference.*

We also provide *a standalone search method* which does all of the above in a single line, and returns the `AutoMLSearch` instance and data check results. If there were data check errors, AutoML will not be run and no `AutoMLSearch` instance will be returned.

### Detecting Problem Type

EvalML includes a simple method, `detect_problem_type`, to help determine the problem type given the target data.

This function can return the predicted problem type as a ProblemType enum, choosing from ProblemType.BINARY, ProblemType.MULTICLASS, and ProblemType.REGRESSION. If the target data is invalid (for instance when there is only 1 unique label), the function will throw an error instead.

[6]:
```
import pandas as pd
from evalml.problem_types import detect_problem_type

y_binary = pd.Series([0, 1, 1, 0, 1, 1])
detect_problem_type(y_binary)
```

[6]:
```
<ProblemTypes.BINARY: 'binary'>
```

### Objective parameter

AutoMLSearch takes in an `objective` parameter to determine which `objective` to optimize for. By default, this parameter is set to `auto`, which allows AutoML to choose `LogLossBinary` for binary classification problems, `LogLossMulticlass` for multiclass classification problems, and `R2` for regression problems.

It should be noted that the `objective` parameter is only used in ranking and helping choose the pipelines to iterate over, but is not used to optimize each individual pipeline during fit-time.

To get the default objective for each problem type, you can use the `get_default_primary_search_objective` function.

[7]:
```
from evalml.automl import get_default_primary_search_objective

binary_objective = get_default_primary_search_objective("binary")
multiclass_objective = get_default_primary_search_objective("multiclass")
regression_objective = get_default_primary_search_objective("regression")

print(binary_objective.name)
```

(continues on next page)

(continued from previous page)

```
print(multiclass_objective.name)
print(regression_objective.name)
```

```
Log Loss Binary
Log Loss Multiclass
R2
```

### Using custom pipelines

EvalML's AutoML algorithm generates a set of pipelines to search with. To provide a custom set instead, set allowed_component_graphs to a dictionary of custom component graphs. `AutoMLSearch` will use these to generate `Pipeline` instances. Note: this will prevent AutoML from generating other pipelines to search over.

[8]:
```
from evalml.pipelines import MulticlassClassificationPipeline


automl_custom = evalml.automl.AutoMLSearch(
    X_train=X_train,
    y_train=y_train,
    problem_type="multiclass",
    verbose=True,
    allowed_component_graphs={
        "My_pipeline": ["Simple Imputer", "Random Forest Classifier"],
        "My_other_pipeline": ["One Hot Encoder", "Random Forest Classifier"],
    },
)
```

```
AutoMLSearch will use mean CV score to rank pipelines.
Removing columns ['currency'] because they are of 'Unknown' type
Using default limit of max_batches=3.
```

### Stopping the search early

To stop the search early, hit `Ctrl-C`. This will bring up a prompt asking for confirmation. Responding with `y` will immediately stop the search. Responding with `n` will continue the search.

### Callback functions

`AutoMLSearch` supports several callback functions, which can be specified as parameters when initializing an `AutoMLSearch` object. They are:

- `start_iteration_callback`
- `add_result_callback`
- `error_callback`

### Start Iteration Callback

Users can set `start_iteration_callback` to set what function is called before each pipeline training iteration. This callback function must take three positional parameters: the pipeline class, the pipeline parameters, and the `AutoMLSearch` object.

[9]:
```
## start_iteration_callback example function
def start_iteration_callback_example(pipeline_class, pipeline_params, automl_obj):
    print("Training pipeline with the following parameters:", pipeline_params)
```

### Add Result Callback

Users can set `add_result_callback` to set what function is called after each pipeline training iteration. This callback function must take three positional parameters: a dictionary containing the training results for the new pipeline, an untrained_pipeline containing the parameters used during training, and the `AutoMLSearch` object.

[10]:
```
## add_result_callback example function
def add_result_callback_example(pipeline_results_dict, untrained_pipeline, automl_obj):
    print(
        "Results for trained pipeline with the following parameters:",
        pipeline_results_dict,
    )
```

### Error Callback

Users can set the `error_callback` to set what function called when `search()` errors and raises an `Exception`. This callback function takes three positional parameters: the `Exception raised`, the traceback, and the `AutoMLSearch object`. This callback function must also accept `kwargs`, so `AutoMLSearch` is able to pass along other parameters used by default.

Evalml defines several error callback functions, which can be found under `evalml.automl.callbacks`. They are:

- `silent_error_callback`
- `raise_error_callback`
- `log_and_save_error_callback`
- `raise_and_save_error_callback`
- `log_error_callback` (default used when `error_callback` is None)

[11]:
```
# error_callback example; this is implemented in the evalml library
def raise_error_callback(exception, traceback, automl, **kwargs):
    """Raises the exception thrown by the AutoMLSearch object. Also logs the exception␣
↪as an error."""
    logger.error(f"AutoMLSearch raised a fatal exception: {str(exception)}")
    logger.error("\n".join(traceback))
    raise exception
```

### 4.1.3 View Rankings

A summary of all the pipelines built can be returned as a pandas DataFrame which is sorted by the validation score.

- For AutoML searches completed with a holdout set, the validation score is the holdout score of the pipeline fitted using the entire training dataset.
- For AutoML searches completed without a holdout set, the validation score is the average score across all cross-validation folds.

[12]:
```
automl.rankings
```

[12]:
```
   id                                     pipeline_name  search_order  \
0   8  XGBoost Classifier w/ Label Encoder + Select C...             8
1   4  LightGBM Classifier w/ Label Encoder + Select ...             4
2   2  Random Forest Classifier w/ Label Encoder + Dr...             2
3   1  Random Forest Classifier w/ Label Encoder + Dr...             1
4   5  Extra Trees Classifier w/ Label Encoder + Sele...             5
5   6  Elastic Net Classifier w/ Label Encoder + Sele...             6
6   9  Logistic Regression Classifier w/ Label Encode...             9
7   7  CatBoost Classifier w/ Label Encoder + Select ...             7
8   3  Decision Tree Classifier w/ Label Encoder + Se...             3
9   0       Mode Baseline Binary Classification Pipeline             0

   ranking_score  holdout_score  mean_cv_score  standard_deviation_cv_score  \
0       0.142417       0.142417       0.256950                     0.137180
1       0.160955       0.160955       0.299971                     0.206176
2       0.219145       0.219145       0.254382                     0.045124
3       0.224440       0.224440       0.259030                     0.039520
4       0.348408       0.348408       0.361341                     0.021758
5       0.400375       0.400375       0.374725                     0.050027
6       0.401581       0.401581       0.374364                     0.049925
7       0.556655       0.556655       0.569010                     0.013825
8       1.002879       1.002879       1.448532                     0.696497
9       4.990660       4.990660       4.921248                     0.112910

   percent_better_than_baseline  high_variance_cv  \
0                     94.778757             False
1                     93.904575             False
2                     94.830946             False
3                     94.736507             False
4                     92.657543             False
5                     92.385573             False
6                     92.392914             False
7                     88.437683             False
8                     70.565771              True
9                      0.000000             False

                                        parameters
0  {'Label Encoder': {'positive_label': None}, 'N...
1  {'Label Encoder': {'positive_label': None}, 'N...
2  {'Label Encoder': {'positive_label': None}, 'D...
3  {'Label Encoder': {'positive_label': None}, 'D...
4  {'Label Encoder': {'positive_label': None}, 'N...
5  {'Label Encoder': {'positive_label': None}, 'N...
```

(continues on next page)

(continued from previous page)

```
6  {'Label Encoder': {'positive_label': None}, 'N...
7  {'Label Encoder': {'positive_label': None}, 'N...
8  {'Label Encoder': {'positive_label': None}, 'N...
9  {'Label Encoder': {'positive_label': None}, 'B...
```

### 4.1.4 Describe Pipeline

Each pipeline is given an `id`. We can get more information about any particular pipeline using that `id`. Here, we will get more information about the pipeline with `id = 1`.

[13]:
```
automl.describe_pipeline(1)
```

```
*****************************************************************************************************
* Random Forest Classifier w/ Label Encoder + Drop Columns Transformer + DateTime␣
↪Featurizer + Imputer + One Hot Encoder + Oversampler *
*****************************************************************************************************

Problem Type: binary
Model Family: Random Forest

Pipeline Steps
==============
1. Label Encoder
         * positive_label : None
2. Drop Columns Transformer
         * columns : ['currency']
3. DateTime Featurizer
         * features_to_extract : ['year', 'month', 'day_of_week', 'hour']
         * encode_as_categories : False
         * time_index : None
4. Imputer
         * categorical_impute_strategy : most_frequent
         * numeric_impute_strategy : mean
         * boolean_impute_strategy : most_frequent
         * categorical_fill_value : None
         * numeric_fill_value : None
         * boolean_fill_value : None
5. One Hot Encoder
         * top_n : 10
         * features_to_encode : None
         * categories : None
         * drop : if_binary
         * handle_unknown : ignore
         * handle_missing : error
6. Oversampler
         * sampling_ratio : 0.25
         * k_neighbors_default : 5
         * n_jobs : -1
         * sampling_ratio_dict : None
         * categorical_features : [3, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22,
↪ 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43,␣
↪44, 45, 46, 47, 48, 49]
```

(continues on next page)

(continued from previous page)

```
        * k_neighbors : 5
7. Random Forest Classifier
        * n_estimators : 100
        * max_depth : 6
        * n_jobs : -1

Training
========
Training for binary problems.
Total training time (including CV): 4.3 seconds

Cross Validation
----------------
             Log Loss Binary  MCC Binary  Gini   AUC  Precision    F1  Balanced Accuracy␣
↪Binary  Accuracy Binary # Training # Validation
0                      0.239       0.823 0.841 0.920      1.000 0.829                    ␣
↪ 0.854            0.960        346          174
1                      0.305       0.481 0.528 0.764      0.875 0.452                    ␣
↪ 0.649            0.902        347          173
2                      0.234       0.875 0.848 0.924      1.000 0.884                    ␣
↪ 0.896            0.971        347          173
mean                   0.259       0.726 0.739 0.869      0.958 0.722                    ␣
↪ 0.800            0.944          -            -
std                    0.040       0.214 0.183 0.091      0.072 0.235                    ␣
↪ 0.132            0.037          -            -
coef of var            0.153       0.294 0.247 0.105      0.075 0.326                    ␣
↪ 0.165            0.039          -            -
```

### 4.1.5 Get Pipeline

We can get the object of any pipeline via their `id` as well:

[14]:
```
pipeline = automl.get_pipeline(1)
print(pipeline.name)
print(pipeline.parameters)
```

```
Random Forest Classifier w/ Label Encoder + Drop Columns Transformer + DateTime␣
↪Featurizer + Imputer + One Hot Encoder + Oversampler
{'Label Encoder': {'positive_label': None}, 'Drop Columns Transformer': {'columns': [
↪'currency']}, 'DateTime Featurizer': {'features_to_extract': ['year', 'month', 'day_of_
↪week', 'hour'], 'encode_as_categories': False, 'time_index': None}, 'Imputer': {
↪'categorical_impute_strategy': 'most_frequent', 'numeric_impute_strategy': 'mean',
↪'boolean_impute_strategy': 'most_frequent', 'categorical_fill_value': None, 'numeric_
↪fill_value': None, 'boolean_fill_value': None}, 'One Hot Encoder': {'top_n': 10,
↪'features_to_encode': None, 'categories': None, 'drop': 'if_binary', 'handle_unknown':
↪'ignore', 'handle_missing': 'error'}, 'Oversampler': {'sampling_ratio': 0.25, 'k_
↪neighbors_default': 5, 'n_jobs': -1, 'sampling_ratio_dict': None, 'categorical_features
↪': [3, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29,␣
↪30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49], 'k_
↪neighbors': 5}, 'Random Forest Classifier': {'n_estimators': 100, 'max_depth': 6, 'n_
↪jobs': -1}}
```

### Get best pipeline

If you specifically want to get the best pipeline, there is a convenient accessor for that. The pipeline returned is already fitted on the input X, y data that we passed to AutoMLSearch. To turn off this default behavior, set `train_best_pipeline=False` when initializing AutoMLSearch.

[15]:
```
best_pipeline = automl.best_pipeline
print(best_pipeline.name)
print(best_pipeline.parameters)
best_pipeline.predict(X_train)
```

```
XGBoost Classifier w/ Label Encoder + Select Columns By Type Transformer + Label Encoder␣
↪+ Drop Columns Transformer + DateTime Featurizer + Imputer + Select Columns␣
↪Transformer + Select Columns Transformer + Label Encoder + Imputer + One Hot Encoder +␣
↪Oversampler
{'Label Encoder': {'positive_label': None}, 'Numeric Pipeline - Select Columns By Type␣
↪Transformer': {'column_types': ['category', 'EmailAddress', 'URL'], 'exclude': True},
↪'Numeric Pipeline - Label Encoder': {'positive_label': None}, 'Numeric Pipeline - Drop␣
↪Columns Transformer': {'columns': ['currency']}, 'Numeric Pipeline - DateTime␣
↪Featurizer': {'features_to_extract': ['year', 'month', 'day_of_week', 'hour'], 'encode_
↪as_categories': False, 'time_index': None}, 'Numeric Pipeline - Imputer': {
↪'categorical_impute_strategy': 'most_frequent', 'numeric_impute_strategy': 'mean',
↪'boolean_impute_strategy': 'most_frequent', 'categorical_fill_value': None, 'numeric_
↪fill_value': None, 'boolean_fill_value': None}, 'Numeric Pipeline - Select Columns␣
↪Transformer': {'columns': ['card_id', 'store_id', 'amount', 'customer_present', 'lat',
↪'lng', 'datetime_month', 'datetime_day_of_week', 'datetime_hour']}, 'Categorical␣
↪Pipeline - Select Columns Transformer': {'columns': ['expiration_date', 'provider',
↪'region', 'country']}, 'Categorical Pipeline - Label Encoder': {'positive_label': None}
↪, 'Categorical Pipeline - Imputer': {'categorical_impute_strategy': 'most_frequent',
↪'numeric_impute_strategy': 'mean', 'boolean_impute_strategy': 'most_frequent',
↪'categorical_fill_value': None, 'numeric_fill_value': None, 'boolean_fill_value': None}
↪, 'Categorical Pipeline - One Hot Encoder': {'top_n': 10, 'features_to_encode': None,
↪'categories': None, 'drop': 'if_binary', 'handle_unknown': 'ignore', 'handle_missing':
↪'error'}, 'Oversampler': {'sampling_ratio': 0.25, 'k_neighbors_default': 5, 'n_jobs': -
↪1, 'sampling_ratio_dict': None, 'categorical_features': [3, 9, 10, 11, 12, 13, 14, 15,␣
↪16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37,
↪ 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48], 'k_neighbors': 5}, 'XGBoost Classifier':
↪{'eta': 0.1, 'max_depth': 6, 'min_child_weight': 1, 'n_estimators': 100, 'n_jobs': -1,
↪'eval_metric': 'logloss'}}
```

[15]:
```
id
144    False
253     True
221    False
432    False
384    False
       ...
128    False
98     False
472    False
642    False
494    False
Name: fraud, Length: 520, dtype: bool
```

### 4.1.6 Training and Scoring Multiple Pipelines using AutoMLSearch

AutoMLSearch will automatically fit the best pipeline on the entire training data. It also provides an easy API for training and scoring other pipelines.

If you'd like to train one or more pipelines on the entire training data, you can use the `train_pipelines` method.

Similarly, if you'd like to score one or more pipelines on a particular dataset, you can use the `score_pipelines` method.

[16]:
```
trained_pipelines = automl.train_pipelines([automl.get_pipeline(i) for i in [0, 1, 2]])
trained_pipelines
```

[16]:
```
{'Mode Baseline Binary Classification Pipeline': pipeline =␣
↪BinaryClassificationPipeline(component_graph={'Label Encoder': ['Label Encoder', 'X',
↪'y'], 'Baseline Classifier': ['Baseline Classifier', 'Label Encoder.x', 'Label Encoder.
↪y']}, parameters={'Label Encoder':{'positive_label': None}, 'Baseline Classifier':{
↪'strategy': 'mode'}}, custom_name='Mode Baseline Binary Classification Pipeline',␣
↪random_seed=0),
 'Random Forest Classifier w/ Label Encoder + Drop Columns Transformer + DateTime␣
↪Featurizer + Imputer + One Hot Encoder + Oversampler': pipeline =␣
↪BinaryClassificationPipeline(component_graph={'Label Encoder': ['Label Encoder', 'X',
↪'y'], 'Drop Columns Transformer': ['Drop Columns Transformer', 'X', 'Label Encoder.y'],
↪ 'DateTime Featurizer': ['DateTime Featurizer', 'Drop Columns Transformer.x', 'Label␣
↪Encoder.y'], 'Imputer': ['Imputer', 'DateTime Featurizer.x', 'Label Encoder.y'], 'One␣
↪Hot Encoder': ['One Hot Encoder', 'Imputer.x', 'Label Encoder.y'], 'Oversampler': [
↪'Oversampler', 'One Hot Encoder.x', 'Label Encoder.y'], 'Random Forest Classifier': [
↪'Random Forest Classifier', 'Oversampler.x', 'Oversampler.y']}, parameters={'Label␣
↪Encoder':{'positive_label': None}, 'Drop Columns Transformer':{'columns': ['currency']}
↪, 'DateTime Featurizer':{'features_to_extract': ['year', 'month', 'day_of_week', 'hour
↪'], 'encode_as_categories': False, 'time_index': None}, 'Imputer':{'categorical_impute_
↪strategy': 'most_frequent', 'numeric_impute_strategy': 'mean', 'boolean_impute_strategy
↪': 'most_frequent', 'categorical_fill_value': None, 'numeric_fill_value': None,
↪'boolean_fill_value': None}, 'One Hot Encoder':{'top_n': 10, 'features_to_encode':␣
↪None, 'categories': None, 'drop': 'if_binary', 'handle_unknown': 'ignore', 'handle_
↪missing': 'error'}, 'Oversampler':{'sampling_ratio': 0.25, 'k_neighbors_default': 5,
↪'n_jobs': -1, 'sampling_ratio_dict': None, 'categorical_features': [3, 10, 11, 12, 13,␣
↪14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35,
↪ 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49], 'k_neighbors': 5}, 'Random␣
↪Forest Classifier':{'n_estimators': 100, 'max_depth': 6, 'n_jobs': -1}}, random_
↪seed=0),
 'Random Forest Classifier w/ Label Encoder + Drop Columns Transformer + DateTime␣
↪Featurizer + Imputer + One Hot Encoder + Oversampler + RF Classifier Select From Model
↪': pipeline = BinaryClassificationPipeline(component_graph={'Label Encoder': ['Label␣
↪Encoder', 'X', 'y'], 'Drop Columns Transformer': ['Drop Columns Transformer', 'X',
↪'Label Encoder.y'], 'DateTime Featurizer': ['DateTime Featurizer', 'Drop Columns␣
↪Transformer.x', 'Label Encoder.y'], 'Imputer': ['Imputer', 'DateTime Featurizer.x',
↪'Label Encoder.y'], 'One Hot Encoder': ['One Hot Encoder', 'Imputer.x', 'Label Encoder.
↪y'], 'Oversampler': ['Oversampler', 'One Hot Encoder.x', 'Label Encoder.y'], 'RF␣
↪Classifier Select From Model': ['RF Classifier Select From Model', 'Oversampler.x',
↪'Oversampler.y'], 'Random Forest Classifier': ['Random Forest Classifier', 'RF␣
↪Classifier Select From Model.x', 'Oversampler.y']}, parameters={'Label Encoder':{
↪'positive_label': None}, 'Drop Columns Transformer':{'columns': ['currency']},
↪'DateTime Featurizer':{'features_to_extract': ['year', 'month', 'day_of_week', 'hour'],
↪ 'encode_as_categories': False, 'time_index': None}, 'Imputer':{'categorical_impute_
↪strategy': 'most_frequent', 'numeric_impute_strategy': 'mean', 'boolean_impute_strategy
↪': 'most_frequent', 'categorical_fill_value': None, 'numeric_fill_value': None,
↪'boolean_fill_value': None}, 'One Hot Encoder':{'top_n': 10, 'features_to_encode':␣
↪None, 'categories': None, 'drop': 'if_binary', 'handle_unknown': 'ignore', 'handle_
↪missing': 'error'}, 'Oversampler':{'sampling_ratio': 0.25, 'k_neighbors_default': 5,
↪'n_jobs': -1, 'sampling_ratio_dict': None, 'categorical_features': [3, 10, 11, 12, 13,␣
↪14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35,
```

(continued from previous page)

[17]:
```
pipeline_holdout_scores = automl.score_pipelines(
    [trained_pipelines[name] for name in trained_pipelines.keys()],
    X_holdout,
    y_holdout,
    ["Accuracy Binary", "F1", "AUC"],
)
pipeline_holdout_scores
```

[17]:
```
{'Mode Baseline Binary Classification Pipeline': OrderedDict([('Accuracy Binary',
               0.8615384615384616),
              ('F1', 0.0),
              ('AUC', 0.5)]),
 'Random Forest Classifier w/ Label Encoder + Drop Columns Transformer + DateTime␣
↪Featurizer + Imputer + One Hot Encoder + Oversampler': OrderedDict([('Accuracy Binary',
               0.9615384615384616),
              ('F1', 0.8387096774193548),
              ('AUC', 0.9265873015873015)]),
 'Random Forest Classifier w/ Label Encoder + Drop Columns Transformer + DateTime␣
↪Featurizer + Imputer + One Hot Encoder + Oversampler + RF Classifier Select From Model
↪': OrderedDict([('Accuracy Binary',
               0.9615384615384616),
              ('F1', 0.8387096774193548),
              ('AUC', 0.9122023809523809)])}
```

### 4.1.7 Saving AutoMLSearch and pipelines from AutoMLSearch

There are two ways to save results from AutoMLSearch.

- You can save the AutoMLSearch object itself, calling `.save(<filepath>)` to do so. This will allow you to save the AutoMLSearch state and reload all pipelines from this.
- If you want to save a pipeline from AutoMLSearch for future use, pipeline classes themselves have a `.save(<filepath>)` method.

[18]:
```
# saving the entire automl search
automl.save("automl.cloudpickle")
automl2 = evalml.automl.AutoMLSearch.load("automl.cloudpickle")
# saving the best pipeline using .save()
best_pipeline.save("pipeline.cloudpickle")
best_pipeline_copy = evalml.pipelines.PipelineBase.load("pipeline.cloudpickle")
```

### 4.1.8 Limiting the AutoML Search Space

The AutoML search algorithm first trains each component in the pipeline with their default values. After the first iteration, it then tweaks the parameters of these components using the pre-defined hyperparameter ranges that these components have. To limit the search over certain hyperparameter ranges, you can specify a `search_parameters` argument with your `AutoMLSearch` parameters. These parameters will limit the hyperparameter search space or pipeline parameter space.

Hyperparameter ranges can be found through the API reference for each component. Parameter arguments must be specified as dictionaries, but the associated values must be `skopt.space` Real, Integer, Categorical objects for setting hyperparameter ranges.

If however you'd like to specify certain values for the initial batch of the AutoML search algorithm, you can use the `search_parameters` argument with non `skopt.space` objects. This will set the initial batch's component parameters to the values passed by this argument.

[19]:
```
from evalml import AutoMLSearch
from evalml.demos import load_fraud
from skopt.space import Categorical
from evalml.model_family import ModelFamily
import woodwork as ww

X, y = load_fraud(n_rows=1000)

# example of setting parameter to just one value
search_parameters = {"Imputer": {"numeric_impute_strategy": "mean"}}


# limit the numeric impute strategy to include only `median` and `most_frequent`
# `mean` is the default value for this argument, but it doesn't need to be included in␣
↪the specified hyperparameter range for this to work
search_parameters = {
    "Imputer": {"numeric_impute_strategy": Categorical(["median", "most_frequent"])}
}

# using this custom hyperparameter means that our Imputer components in these pipelines␣
↪will only search through
# 'median' and 'most_frequent' strategies for 'numeric_impute_strategy'
automl_constrained = AutoMLSearch(
    X_train=X,
    y_train=y,
    problem_type="binary",
    search_parameters=search_parameters,
    verbose=True,
)
```

```
             Number of Features
Boolean                       1
Categorical                   6
Numeric                       5

Number of training examples: 1000
Targets
False    85.90%
True     14.10%
```

(continues on next page)

(continued from previous page)

```
Name: fraud, dtype: object
AutoMLSearch will use mean CV score to rank pipelines.
Using default limit of max_batches=3.
```

A `skopt.space` Integer, Real, or Categorical will set the hyperparameter space explored during search. All other values will set the pipeline parameters directly. Setting pipeline parameters directly defines the initialization parameters that a pipeline starts with during the first batch of AutoMLSearch. the hyperparameter range then defines the space of possible new parameter values, which the tuner chooses.

Let's walk through some examples to explain this. For instance,

```
search_parameters = {'Imputer': {
    'numeric_impute_strategy': 'mean'
}}
```

then in the initial search, the algorithm would use `mean` as the impute strategy in batch 1. However, since `Imputer.numeric_impute_strategy` has a valid hyperparameter range, if the algorithm suggests a different strategy, it can and will change this value. To limit this to using `mean` only for the duration of the search, it is necessary to use the `skopt.space`:

```
search_parameters = {'Imputer': {
    'numeric_impute_strategy': Categorical(['mean'])
}}
```

However, if a value has no hyperparameter range associated, then the algorithm will use this value as the only parameter. For instance,

```
search_parameters = {'Label Encoder': {
    'positive_label': True
}}
```

Since `Label Encoder.positive_label` has no associated hyperparameter range, the algorithm will use this parameter for the entire duration of the search.

### 4.1.9 Imbalanced Data

The AutoML search algorithm now has functionality to handle imbalanced data during classification! AutoMLSearch now provides two additional parameters, `sampler_method` and `sampler_balanced_ratio`, that allow you to let AutoMLSearch know whether to sample imbalanced data, and how to do so. `sampler_method` takes in either `Undersampler`, `Oversampler`, `auto`, or None as the sampler to use, and `sampler_balanced_ratio` specifies the `minority/majority` ratio that you want to sample to. Details on the Undersampler and Oversampler components can be found in the documentation.

This can be used for imbalanced datasets, like the fraud dataset, which has a 'minority:majority' ratio of < 0.2.

[20]:
```
automl_auto = AutoMLSearch(
    X_train=X, y_train=y, problem_type="binary", automl_algorithm="iterative"
)
automl_auto.allowed_pipelines[-1]
```

[20]:
```
pipeline = BinaryClassificationPipeline(component_graph={'Label Encoder': ['Label Encoder
↪', 'X', 'y'], 'DateTime Featurizer': ['DateTime Featurizer', 'X', 'Label Encoder.y'],
↪'Imputer': ['Imputer', 'DateTime Featurizer.x', 'Label Encoder.y'], 'One Hot Encoder':␣
↪['One Hot Encoder', 'Imputer.x', 'Label Encoder.y'], 'Oversampler': ['Oversampler',
↪'One Hot Encoder.x', 'Label Encoder.y'], 'Extra Trees Classifier': ['Extra Trees␣
↪Classifier', 'Oversampler.x', 'Oversampler.y']}, parameters={'Label Encoder':{
↪'positive_label': None}, 'DateTime Featurizer':{'features_to_extract': ['year', 'month
↪', 'day_of_week', 'hour'], 'encode_as_categories': False, 'time_index': None}, 'Imputer
↪':{'categorical_impute_strategy': 'most_frequent', 'numeric_impute_strategy': 'mean',
↪'boolean_impute_strategy': 'most_frequent', 'categorical_fill_value': None, 'numeric
```

(continues on next page)

(continued from previous page)

The Oversampler is chosen as the default sampling component here, since the `sampler_balanced_ratio = 0.25`. If you specified a lower ratio, for instance `sampler_balanced_ratio = 0.1`, then there would be no sampling component added here. This is because if a ratio of 0.1 would be considered balanced, then a ratio of 0.2 would also be balanced.

The Oversampler uses SMOTE under the hood, and automatically selects whether to use SMOTE, SMOTEN, or SMOTENC based on the data it receives.

[21]:
```
automl_auto_ratio = AutoMLSearch(
    X_train=X,
    y_train=y,
    problem_type="binary",
    sampler_balanced_ratio=0.1,
    automl_algorithm="iterative",
)
automl_auto_ratio.allowed_pipelines[-1]
```

[21]:
```
pipeline = BinaryClassificationPipeline(component_graph={'Label Encoder': ['Label Encoder
↪', 'X', 'y'], 'DateTime Featurizer': ['DateTime Featurizer', 'X', 'Label Encoder.y'],
↪'Imputer': ['Imputer', 'DateTime Featurizer.x', 'Label Encoder.y'], 'One Hot Encoder':␣
↪['One Hot Encoder', 'Imputer.x', 'Label Encoder.y'], 'Extra Trees Classifier': ['Extra␣
↪Trees Classifier', 'One Hot Encoder.x', 'Label Encoder.y']}, parameters={'Label Encoder
↪':{'positive_label': None}, 'DateTime Featurizer':{'features_to_extract': ['year',
↪'month', 'day_of_week', 'hour'], 'encode_as_categories': False, 'time_index': None},
↪'Imputer':{'categorical_impute_strategy': 'most_frequent', 'numeric_impute_strategy':
↪'mean', 'boolean_impute_strategy': 'most_frequent', 'categorical_fill_value': None,
↪'numeric_fill_value': None, 'boolean_fill_value': None}, 'One Hot Encoder':{'top_n':␣
↪10, 'features_to_encode': None, 'categories': None, 'drop': 'if_binary', 'handle_
↪unknown': 'ignore', 'handle_missing': 'error'}, 'Extra Trees Classifier':{'n_estimators
↪': 100, 'max_features': 'auto', 'max_depth': 6, 'min_samples_split': 2, 'min_weight_
↪fraction_leaf': 0.0, 'n_jobs': -1}}, random_seed=0)
```

Additionally, you can add more fine-grained sampling ratios by passing in a `sampling_ratio_dict` in pipeline parameters. For this dictionary, AutoMLSearch expects the keys to be int values from 0 to `n-1` for the classes, and the values would be the `sampler_balanced__ratio` associated with each target. This dictionary would override the AutoML argument `sampler_balanced_ratio`. Below, you can see the scenario for Oversampler component on this dataset. Note that the logic for Undersamplers is included in the commented section.

[22]:
```
# In this case, the majority class is the negative class
# for the oversampler, we don't want to oversample this class, so class 0 (majority) will␣
↪have a ratio of 1 to itself
# for the minority class 1, we want to oversample it to have a minority/majority ratio␣
↪of 0.5, which means we want minority to have 1/2 the samples as the minority
sampler_ratio_dict = {0: 1, 1: 0.5}
search_parameters = {"Oversampler": {"sampler_balanced_ratio": sampler_ratio_dict}}
automl_auto_ratio_dict = AutoMLSearch(
    X_train=X,
    y_train=y,
    problem_type="binary",
    search_parameters=search_parameters,
    automl_algorithm="iterative",
)
```

(continues on next page)

(continued from previous page)

```
automl_auto_ratio_dict.allowed_pipelines[-1]

# Undersampler case
# we don't want to undersample this class, so class 1 (minority) will have a ratio of 1␣
↪to itself
# for the majority class 0, we want to undersample it to have a minority/majority ratio␣
↪of 0.5, which means we want majority to have 2x the samples as the minority
# sampler_ratio_dict = {0: 0.5, 1: 1}
# search_parameters = {"Oversampler": {"sampler_balanced_ratio": sampler_ratio_dict}}
# automl_auto_ratio_dict = AutoMLSearch(X_train=X, y_train=y, problem_type='binary',␣
↪search_parameters=search_parameters)
```

[22]:
```
pipeline = BinaryClassificationPipeline(component_graph={'Label Encoder': ['Label Encoder
↪', 'X', 'y'], 'DateTime Featurizer': ['DateTime Featurizer', 'X', 'Label Encoder.y'],
↪'Imputer': ['Imputer', 'DateTime Featurizer.x', 'Label Encoder.y'], 'One Hot Encoder':␣
↪['One Hot Encoder', 'Imputer.x', 'Label Encoder.y'], 'Oversampler': ['Oversampler',
↪'One Hot Encoder.x', 'Label Encoder.y'], 'Extra Trees Classifier': ['Extra Trees␣
↪Classifier', 'Oversampler.x', 'Oversampler.y']}, parameters={'Label Encoder':{
↪'positive_label': None}, 'DateTime Featurizer':{'features_to_extract': ['year', 'month
↪', 'day_of_week', 'hour'], 'encode_as_categories': False, 'time_index': None}, 'Imputer
↪':{'categorical_impute_strategy': 'most_frequent', 'numeric_impute_strategy': 'mean',
↪'boolean_impute_strategy': 'most_frequent', 'categorical_fill_value': None, 'numeric_
↪fill_value': None, 'boolean_fill_value': None}, 'One Hot Encoder':{'top_n': 10,
↪'features_to_encode': None, 'categories': None, 'drop': 'if_binary', 'handle_unknown':
↪'ignore', 'handle_missing': 'error'}, 'Oversampler':{'sampling_ratio': 0.25, 'k_
↪neighbors_default': 5, 'n_jobs': -1, 'sampling_ratio_dict': None, 'sampler_balanced_
↪ratio': {0: 1, 1: 0.5}}, 'Extra Trees Classifier':{'n_estimators': 100, 'max_features':
↪ 'auto', 'max_depth': 6, 'min_samples_split': 2, 'min_weight_fraction_leaf': 0.0, 'n_
↪jobs': -1}}, random_seed=0)
```

### 4.1.10 Adding ensemble methods to AutoML

#### Stacking

Stacking is an ensemble machine learning algorithm that involves training a model to best combine the predictions of several base learning algorithms. First, each base learning algorithms is trained using the given data. Then, the combining algorithm or meta-learner is trained on the predictions made by those base learning algorithms to make a final prediction.

AutoML enables stacking using the `ensembling` flag during initalization; this is set to `False` by default. How ensembling runs is defined by the AutoML algorithm you choose. In the `IterativeAlgorithm`, the stacking ensemble pipeline runs in its own batch after a whole cycle of training has occurred (each allowed pipeline trains for one batch). Note that this means **a large number of iterations may need to run before the stacking ensemble runs**. It is also important to note that **only the first CV fold is calculated for stacking ensembles** because the model internally uses CV folds. See below in the AutoML Algorithms section to see how ensembling is run for `DefaultAlgorithm`. Please do note that ensembling is currently unavailable for time series problems.

[23]:
```
X, y = evalml.demos.load_breast_cancer()

automl_with_ensembling = AutoMLSearch(
    X_train=X,
    y_train=y,
```

(continues on next page)

(continued from previous page)

```
    problem_type="binary",
    allowed_model_families=[ModelFamily.LINEAR_MODEL],
    max_batches=4,
    ensembling=True,
    automl_algorithm="iterative",
    verbose=True,
)
automl_with_ensembling.search(interactive_plot=False)
```

```
         Number of Features
Numeric                  30

Number of training examples: 569
Targets
benign       62.74%
malignant    37.26%
Name: target, dtype: object
AutoMLSearch will use mean CV score to rank pipelines.
Generating pipelines to search over...
Ensembling will run every 3 batches.
2 pipelines ready for search.

*****************************
* Beginning pipeline search *
*****************************

Optimizing for Log Loss Binary.
Lower score is better.

Using SequentialEngine to train and score pipelines.
Searching up to 4 batches for a total of 14 pipelines.
Allowed model families: linear_model, linear_model

Evaluating Baseline Pipeline: Mode Baseline Binary Classification Pipeline
Mode Baseline Binary Classification Pipeline:
        Starting cross validation
        Finished cross validation - mean Log Loss Binary: 13.429

*****************************
* Evaluating Batch Number 1 *
*****************************

Elastic Net Classifier w/ Label Encoder + Imputer + Standard Scaler:
        Starting cross validation
        Finished cross validation - mean Log Loss Binary: 0.077
Logistic Regression Classifier w/ Label Encoder + Imputer + Standard Scaler:
        Starting cross validation
        Finished cross validation - mean Log Loss Binary: 0.077

*****************************
* Evaluating Batch Number 2 *
*****************************
```

(continues on next page)

(continued from previous page)

```
Logistic Regression Classifier w/ Label Encoder + Imputer + Standard Scaler:
        Starting cross validation
        Finished cross validation - mean Log Loss Binary: 0.090
Logistic Regression Classifier w/ Label Encoder + Imputer + Standard Scaler:
        Starting cross validation
        Finished cross validation - mean Log Loss Binary: 0.085
Logistic Regression Classifier w/ Label Encoder + Imputer + Standard Scaler:
        Starting cross validation
        Finished cross validation - mean Log Loss Binary: 0.081
Logistic Regression Classifier w/ Label Encoder + Imputer + Standard Scaler:
        Starting cross validation
        Finished cross validation - mean Log Loss Binary: 0.097
Logistic Regression Classifier w/ Label Encoder + Imputer + Standard Scaler:
        Starting cross validation
        Finished cross validation - mean Log Loss Binary: 0.093

*****************************
* Evaluating Batch Number 3 *
*****************************

Elastic Net Classifier w/ Label Encoder + Imputer + Standard Scaler:
        Starting cross validation
        Finished cross validation - mean Log Loss Binary: 0.075
Elastic Net Classifier w/ Label Encoder + Imputer + Standard Scaler:
        Starting cross validation
        Finished cross validation - mean Log Loss Binary: 0.076
Elastic Net Classifier w/ Label Encoder + Imputer + Standard Scaler:
        Starting cross validation
        Finished cross validation - mean Log Loss Binary: 0.075
Elastic Net Classifier w/ Label Encoder + Imputer + Standard Scaler:
        Starting cross validation
        Finished cross validation - mean Log Loss Binary: 0.079
Elastic Net Classifier w/ Label Encoder + Imputer + Standard Scaler:
        Starting cross validation
        Finished cross validation - mean Log Loss Binary: 0.075

*****************************
* Evaluating Batch Number 4 *
*****************************

Stacked Ensemble Classification Pipeline:
        Starting cross validation
        Finished cross validation - mean Log Loss Binary: 0.103

Search finished after 00:20
Best pipeline: Elastic Net Classifier w/ Label Encoder + Imputer + Standard Scaler
Best pipeline Log Loss Binary: 0.075391
```

Data type cannot be displayed: application/vnd.plotly.v1+json, text/html

[23]:
```
{1: {'Elastic Net Classifier w/ Label Encoder + Imputer + Standard Scaler': '00:01',
  'Logistic Regression Classifier w/ Label Encoder + Imputer + Standard Scaler': '00:01',
  'Total time of batch': '00:03'},
 2: {'Logistic Regression Classifier w/ Label Encoder + Imputer + Standard Scaler': '00:
↪01',
  'Total time of batch': '00:07'},
 3: {'Elastic Net Classifier w/ Label Encoder + Imputer + Standard Scaler': '00:01',
  'Total time of batch': '00:07'},
 4: {'Stacked Ensemble Classification Pipeline': '00:01',
  'Total time of batch': '00:01'}}
```

We can view more information about the stacking ensemble pipeline (which was the best performing pipeline) by calling `.describe()`.

[24]:
```
automl_with_ensembling.best_pipeline.describe()
```

```
*********************************************************************
* Elastic Net Classifier w/ Label Encoder + Imputer + Standard Scaler *
*********************************************************************

Problem Type: binary
Model Family: Linear
Number of features: 30

Pipeline Steps
==============
1. Label Encoder
         * positive_label : None
2. Imputer
         * categorical_impute_strategy : most_frequent
         * numeric_impute_strategy : knn
         * boolean_impute_strategy : most_frequent
         * categorical_fill_value : None
         * numeric_fill_value : None
         * boolean_fill_value : None
3. Standard Scaler
4. Elastic Net Classifier
         * penalty : elasticnet
         * C : 8.474044870453413
         * l1_ratio : 0.6235636967859725
         * n_jobs : -1
         * multi_class : auto
         * solver : saga
```

### 4.1.11 AutoML Algorithms

EvalML currently has two algorithms available for users to choose from. Below, we will run through how each algorithm works and how to access them through `AutoMLSearch` as well as the top level search methods.

#### IterativeAlgorithm

`IterativeAlgorithm` is the first AutoML Algorithm created in EvalML and can be acessed with the `search_iterative` method or specifiying `AutoMLSearch(automl_algorithm='iterative')`. The algorithm works as follows:

- Every batch (after the initial baseline model) contains pipelines of all available estimators for the specified problem type
- Pipelines contain preprocessing (imputing, encoding, etc.) needed for machine learning but no feature selection is applied
- Ensembling can be turned on by passing in the `ensembling=True` parameter and will be run after a whole cycle of training has occurred (each allowed pipeline trains for one batch)

[25]:
```
import evalml

X, y = evalml.demos.load_fraud(n_rows=250)
```

```
             Number of Features
Boolean                       1
Categorical                   6
Numeric                       5

Number of training examples: 250
Targets
False    88.40%
True     11.60%
Name: fraud, dtype: object
```

[26]:
```
from evalml.automl import search_iterative

# top level search method will run `AutoMLSearch` with `IterativeAlgorithm` as well as␣
↪apply our default data checks
auto_iterative, messages_iterative = search_iterative(X, y, problem_type="binary")
```

[27]:
```
from evalml import AutoMLSearch

auto_iterative = AutoMLSearch(
    X_train=X,
    y_train=y,
    problem_type="binary",
    automl_algorithm="iterative",
    verbose=True,
)
auto_iterative.search(interactive_plot=False)
```

```
AutoMLSearch will use mean CV score to rank pipelines.
Removing columns ['currency', 'expiration_date'] because they are of 'Unknown' type
```

(continues on next page)

(continued from previous page)

```
Generating pipelines to search over...
8 pipelines ready for search.
Using default limit of max_batches=1.


*****************************
* Beginning pipeline search *
*****************************

Optimizing for Log Loss Binary.
Lower score is better.

Using SequentialEngine to train and score pipelines.
Searching up to 1 batches for a total of None pipelines.
Allowed model families: linear_model, linear_model, xgboost, lightgbm, catboost, random_
↪forest, decision_tree, extra_trees

Evaluating Baseline Pipeline: Mode Baseline Binary Classification Pipeline
Mode Baseline Binary Classification Pipeline:
        Starting cross validation
        Finished cross validation - mean Log Loss Binary: 4.181

*****************************
* Evaluating Batch Number 1 *
*****************************

Elastic Net Classifier w/ Label Encoder + Drop Columns Transformer + DateTime Featurizer␣
↪+ Imputer + One Hot Encoder + Oversampler + Standard Scaler:
        Starting cross validation
        Finished cross validation - mean Log Loss Binary: 0.429
Logistic Regression Classifier w/ Label Encoder + Drop Columns Transformer + DateTime␣
↪Featurizer + Imputer + One Hot Encoder + Oversampler + Standard Scaler:
        Starting cross validation
        Finished cross validation - mean Log Loss Binary: 0.429
XGBoost Classifier w/ Label Encoder + Drop Columns Transformer + DateTime Featurizer +␣
↪Imputer + One Hot Encoder + Oversampler:
        Starting cross validation
        Finished cross validation - mean Log Loss Binary: 0.266
LightGBM Classifier w/ Label Encoder + Drop Columns Transformer + DateTime Featurizer +␣
↪Imputer + One Hot Encoder + Oversampler:
        Starting cross validation
        Finished cross validation - mean Log Loss Binary: 0.325
CatBoost Classifier w/ Label Encoder + Drop Columns Transformer + DateTime Featurizer +␣
↪Imputer + Oversampler:
        Starting cross validation
        Finished cross validation - mean Log Loss Binary: 0.596
Random Forest Classifier w/ Label Encoder + Drop Columns Transformer + DateTime␣
↪Featurizer + Imputer + One Hot Encoder + Oversampler:
        Starting cross validation
        Finished cross validation - mean Log Loss Binary: 0.287
Decision Tree Classifier w/ Label Encoder + Drop Columns Transformer + DateTime␣
↪Featurizer + Imputer + One Hot Encoder + Oversampler:
```

(continues on next page)

(continued from previous page)

```
        Starting cross validation
        Finished cross validation - mean Log Loss Binary: 4.640
        High coefficient of variation (cv >= 0.5) within cross validation scores.
        Decision Tree Classifier w/ Label Encoder + Drop Columns Transformer + DateTime␣
↪Featurizer + Imputer + One Hot Encoder + Oversampler may not perform as estimated on␣
↪unseen data.
Extra Trees Classifier w/ Label Encoder + Drop Columns Transformer + DateTime Featurizer␣
↪+ Imputer + One Hot Encoder + Oversampler:
        Starting cross validation
        Finished cross validation - mean Log Loss Binary: 0.347

Search finished after 00:21
Best pipeline: XGBoost Classifier w/ Label Encoder + Drop Columns Transformer + DateTime␣
↪Featurizer + Imputer + One Hot Encoder + Oversampler
Best pipeline Log Loss Binary: 0.266464
```

Data type cannot be displayed: application/vnd.plotly.v1+json, text/html

[27]:
```
{1: {'Elastic Net Classifier w/ Label Encoder + Drop Columns Transformer + DateTime␣
↪Featurizer + Imputer + One Hot Encoder + Oversampler + Standard Scaler': '00:02',
 'Logistic Regression Classifier w/ Label Encoder + Drop Columns Transformer + DateTime␣
↪Featurizer + Imputer + One Hot Encoder + Oversampler + Standard Scaler': '00:02',
 'XGBoost Classifier w/ Label Encoder + Drop Columns Transformer + DateTime Featurizer␣
↪+ Imputer + One Hot Encoder + Oversampler': '00:02',
 'LightGBM Classifier w/ Label Encoder + Drop Columns Transformer + DateTime Featurizer␣
↪+ Imputer + One Hot Encoder + Oversampler': '00:02',
 'CatBoost Classifier w/ Label Encoder + Drop Columns Transformer + DateTime Featurizer␣
↪+ Imputer + Oversampler': '00:01',
 'Random Forest Classifier w/ Label Encoder + Drop Columns Transformer + DateTime␣
↪Featurizer + Imputer + One Hot Encoder + Oversampler': '00:02',
 'Decision Tree Classifier w/ Label Encoder + Drop Columns Transformer + DateTime␣
↪Featurizer + Imputer + One Hot Encoder + Oversampler': '00:01',
 'Extra Trees Classifier w/ Label Encoder + Drop Columns Transformer + DateTime␣
↪Featurizer + Imputer + One Hot Encoder + Oversampler': '00:03',
 'Total time of batch': '00:20'}}
```

### DefaultAlgorithm

`DefaultAlgorithm` was designed to do three main things:

1. Abstract out more parameters and decisions from the user.
2. Perform deeper tuning for high performing pipelines.
3. Create a platform to introduce feature selection as well as other potential techniques/heuristics for AutoML.

`DefaultAlgorithm` does this by creating the concept of two modes: `fast` and `long`, where `fast` is a subset of long. The algorithm runs as follows:

1. Run naive pipelines:

a. a linear model with the default preprocessing pipeline

b. a random forest pipeline with the default preprocessing pipeline

2. Run the same pipelines, this time with feature selection. Subsequent pipelines will use the selected features with a SelectedColumns transformer.
3. Run all pipelines with preprocessing components:
   a. scan rest of estimators (`IterativeAlgorithm` batch 1).
4. First ensembling run

Fast mode ends here. Begin long mode.

6. Run top 3 estimators:
   a. Generate 50 random parameter sets. Run all 150 in one batch
7. Second ensembling run
8. Repeat 8a and 8b indefinitely until the specified time in `AutoMLSearch` is met:
   a. For each of the previous top 3 estimators, sample 10 parameters from the tuner. Run all 30 in one batch
   b. Run ensembling

To this end, it is recommended to use the top level `search()` method to run `DefaultAlgorithm`. This allows users to specify running search with just the `mode` parameter, where `fast` is recommended for users who want a fast scan at how EvalML pipelines will perform on their problem and where `long` is reserved for a deeper dive into high performing pipelines. If one needs finer control over AutoML parameters, one can also specify `automl_algorithm='default'` using `AutoMLSearch` and it will default to using `fast` mode. However, in this case ensembling will be defined by the `ensembling` flag (if `ensembling=False` the abovementioned ensembling batches will be skipped). Users are welcome to select `max_batches` according to the algorithm above (or other stopping criteria) but should be aware that results may not be optimal if the algorithm does not run for the full length of `fast` mode.

[28]:
```
from evalml.automl import search

# top level search method will run `AutoMLSearch` with `IterativeAlgorithm` as well as␣
↪apply our default data checks
auto_default, messages_default = search(X, y, problem_type="binary", mode="fast")
```

```
        High coefficient of variation (cv >= 0.5) within cross validation scores.
        Decision Tree Classifier w/ Label Encoder + Select Columns By Type Transformer +␣
↪Label Encoder + Drop Columns Transformer + DateTime Featurizer + Imputer + Select␣
↪Columns Transformer + Select Columns Transformer + Label Encoder + Imputer + One Hot␣
↪Encoder + Oversampler may not perform as estimated on unseen data.
```

[29]:
```
from evalml import AutoMLSearch

auto_default = AutoMLSearch(
    X_train=X,
    y_train=y,
    problem_type="binary",
    automl_algorithm="default",
    ensembling=True,
    verbose=True,
)
auto_default.search(interactive_plot=False)
```

```
AutoMLSearch will use mean CV score to rank pipelines.
Removing columns ['currency', 'expiration_date'] because they are of 'Unknown' type
Using default limit of max_batches=4.
```

(continues on next page)

(continued from previous page)

```
*****************************
* Beginning pipeline search *
*****************************

Optimizing for Log Loss Binary.
Lower score is better.

Using SequentialEngine to train and score pipelines.
Searching up to 4 batches for a total of None pipelines.
Allowed model families:

Evaluating Baseline Pipeline: Mode Baseline Binary Classification Pipeline
Mode Baseline Binary Classification Pipeline:
        Starting cross validation
        Finished cross validation - mean Log Loss Binary: 4.181

*****************************
* Evaluating Batch Number 1 *
*****************************

Random Forest Classifier w/ Label Encoder + Drop Columns Transformer + DateTime␣
↪Featurizer + Imputer + One Hot Encoder + Oversampler:
        Starting cross validation
        Finished cross validation - mean Log Loss Binary: 0.287

*****************************
* Evaluating Batch Number 2 *
*****************************

Random Forest Classifier w/ Label Encoder + Drop Columns Transformer + DateTime␣
↪Featurizer + Imputer + One Hot Encoder + Oversampler + RF Classifier Select From Model:
        Starting cross validation
        Finished cross validation - mean Log Loss Binary: 0.282

*****************************
* Evaluating Batch Number 3 *
*****************************

Decision Tree Classifier w/ Label Encoder + Select Columns By Type Transformer + Label␣
↪Encoder + Drop Columns Transformer + DateTime Featurizer + Imputer + Select Columns␣
↪Transformer + Select Columns Transformer + Label Encoder + Imputer + One Hot Encoder +␣
↪Oversampler:
        Starting cross validation
        Finished cross validation - mean Log Loss Binary: 2.363
        High coefficient of variation (cv >= 0.5) within cross validation scores.
        Decision Tree Classifier w/ Label Encoder + Select Columns By Type Transformer +␣
↪Label Encoder + Drop Columns Transformer + DateTime Featurizer + Imputer + Select␣
↪Columns Transformer + Select Columns Transformer + Label Encoder + Imputer + One Hot␣
↪Encoder + Oversampler may not perform as estimated on unseen data.
LightGBM Classifier w/ Label Encoder + Select Columns By Type Transformer + Label␣
↪Encoder + Drop Columns Transformer + DateTime Featurizer + Imputer + Select Columns␣
↪Transformer + Select Columns Transformer + Label Encoder + Imputer + One Hot Encoder +␣
↪Oversampler:
```

(continues on next page)

(continued from previous page)

```
        Starting cross validation
        Finished cross validation - mean Log Loss Binary: 0.325
Extra Trees Classifier w/ Label Encoder + Select Columns By Type Transformer + Label␣
↪Encoder + Drop Columns Transformer + DateTime Featurizer + Imputer + Select Columns␣
↪Transformer + Select Columns Transformer + Label Encoder + Imputer + One Hot Encoder +␣
↪Oversampler:
        Starting cross validation
        Finished cross validation - mean Log Loss Binary: 0.348
Elastic Net Classifier w/ Label Encoder + Select Columns By Type Transformer + Label␣
↪Encoder + Drop Columns Transformer + DateTime Featurizer + Imputer + Standard Scaler +␣
↪Select Columns Transformer + Select Columns Transformer + Label Encoder + Imputer +␣
↪One Hot Encoder + Standard Scaler + Oversampler:
        Starting cross validation
        Finished cross validation - mean Log Loss Binary: 0.422
CatBoost Classifier w/ Label Encoder + Select Columns By Type Transformer + Label␣
↪Encoder + Drop Columns Transformer + DateTime Featurizer + Imputer + Select Columns␣
↪Transformer + Select Columns Transformer + Label Encoder + Imputer + Oversampler:
        Starting cross validation
        Finished cross validation - mean Log Loss Binary: 0.596
XGBoost Classifier w/ Label Encoder + Select Columns By Type Transformer + Label Encoder␣
↪+ Drop Columns Transformer + DateTime Featurizer + Imputer + Select Columns␣
↪Transformer + Select Columns Transformer + Label Encoder + Imputer + One Hot Encoder +␣
↪Oversampler:
        Starting cross validation
        Finished cross validation - mean Log Loss Binary: 0.266
Logistic Regression Classifier w/ Label Encoder + Select Columns By Type Transformer +␣
↪Label Encoder + Drop Columns Transformer + DateTime Featurizer + Imputer + Standard␣
↪Scaler + Select Columns Transformer + Select Columns Transformer + Label Encoder +␣
↪Imputer + One Hot Encoder + Standard Scaler + Oversampler:
        Starting cross validation
        Finished cross validation - mean Log Loss Binary: 0.422

*****************************
* Evaluating Batch Number 4 *
*****************************

Stacked Ensemble Classification Pipeline:
        Starting cross validation
        Finished cross validation - mean Log Loss Binary: 0.237

Search finished after 00:37
Best pipeline: Stacked Ensemble Classification Pipeline
Best pipeline Log Loss Binary: 0.236593
```

Data type cannot be displayed: application/vnd.plotly.v1+json, text/html

[29]:
```
{1: {'Random Forest Classifier w/ Label Encoder + Drop Columns Transformer + DateTime␣
↪Featurizer + Imputer + One Hot Encoder + Oversampler': '00:02',
  'Total time of batch': '00:02'},
 2: {'Random Forest Classifier w/ Label Encoder + Drop Columns Transformer + DateTime␣
↪Featurizer + Imputer + One Hot Encoder + Oversampler + RF Classifier Select From Model
↪': '00:03',
```

(continues on next page)

(continued from previous page)

```
 'Total time of batch': '00:03'},
3: {'Decision Tree Classifier w/ Label Encoder + Select Columns By Type Transformer +␣
↪Label Encoder + Drop Columns Transformer + DateTime Featurizer + Imputer + Select␣
↪Columns Transformer + Select Columns Transformer + Label Encoder + Imputer + One Hot␣
↪Encoder + Oversampler': '00:02',
 'LightGBM Classifier w/ Label Encoder + Select Columns By Type Transformer + Label␣
↪Encoder + Drop Columns Transformer + DateTime Featurizer + Imputer + Select Columns␣
↪Transformer + Select Columns Transformer + Label Encoder + Imputer + One Hot Encoder +␣
↪Oversampler': '00:02',
 'Extra Trees Classifier w/ Label Encoder + Select Columns By Type Transformer + Label␣
↪Encoder + Drop Columns Transformer + DateTime Featurizer + Imputer + Select Columns␣
↪Transformer + Select Columns Transformer + Label Encoder + Imputer + One Hot Encoder +␣
↪Oversampler': '00:03',
 'Elastic Net Classifier w/ Label Encoder + Select Columns By Type Transformer + Label␣
↪Encoder + Drop Columns Transformer + DateTime Featurizer + Imputer + Standard Scaler +␣
↪Select Columns Transformer + Select Columns Transformer + Label Encoder + Imputer +␣
↪One Hot Encoder + Standard Scaler + Oversampler': '00:02',
 'CatBoost Classifier w/ Label Encoder + Select Columns By Type Transformer + Label␣
↪Encoder + Drop Columns Transformer + DateTime Featurizer + Imputer + Select Columns␣
↪Transformer + Select Columns Transformer + Label Encoder + Imputer + Oversampler': '00:
↪02',
 'XGBoost Classifier w/ Label Encoder + Select Columns By Type Transformer + Label␣
↪Encoder + Drop Columns Transformer + DateTime Featurizer + Imputer + Select Columns␣
↪Transformer + Select Columns Transformer + Label Encoder + Imputer + One Hot Encoder +␣
↪Oversampler': '00:02',
 'Logistic Regression Classifier w/ Label Encoder + Select Columns By Type Transformer␣
↪+ Label Encoder + Drop Columns Transformer + DateTime Featurizer + Imputer + Standard␣
↪Scaler + Select Columns Transformer + Select Columns Transformer + Label Encoder +␣
↪Imputer + One Hot Encoder + Standard Scaler + Oversampler': '00:02',
 'Total time of batch': '00:19'},
4: {'Stacked Ensemble Classification Pipeline': '00:11',
 'Total time of batch': '00:11'}}
```

### 4.1.12 Pipeline differences

Through the search output above, we can see how pipelines differ between `IterativeAlgorithm` and `DefaultAlgorithm`. This is because `DefaultAlgorithm` utilizes new components such as `RFRegressorSelectFromModel` and other column selectors for feature selection as well as a new pipeline structure to handle feature selection for categorical and non-categorical features.

[30]:
```
auto_iterative.get_pipeline(4).graph()
```

[30]:

[31]:
```
auto_default.get_pipeline(6).graph()
```

[31]:

### 4.1.13 Access raw results

The `AutoMLSearch` class records detailed results information under the `results` field, including information about the cross-validation scoring and parameters.

[32]:
```
import pprint

pp = pprint.PrettyPrinter(indent=0, width=100, depth=3, compact=True, sort_dicts=False)

pp.pprint(automl.results)
```

```
{'pipeline_results': {0: {'id': 0,
                          'pipeline_name': 'Mode Baseline Binary Classification Pipeline',
                          'pipeline_class': <class 'evalml.pipelines.binary_classification_
↪pipeline.BinaryClassificationPipeline'>,
                          'pipeline_summary': 'Baseline Classifier w/ Label Encoder',
                          'parameters': {...},
                          'mean_cv_score': 4.921248270190403,
                          'standard_deviation_cv_score': 0.11291020093698304,
                          'high_variance_cv': False,
                          'training_time': 0.5921878814697266,
                          'cv_data': [...],
                          'percent_better_than_baseline_all_objectives': {...},
                          'percent_better_than_baseline': 0,
                          'ranking_score': 4.990659700031606,
                          'ranking_additional_objectives': {...},
                          'holdout_score': 4.990659700031606},
                      1: {'id': 1,
                          'pipeline_name': 'Random Forest Classifier w/ Label Encoder +␣
↪Drop Columns '
                                            'Transformer + DateTime Featurizer + Imputer +␣
↪One Hot '
                                            'Encoder + Oversampler',
                          'pipeline_class': <class 'evalml.pipelines.binary_classification_
↪pipeline.BinaryClassificationPipeline'>,
                          'pipeline_summary': 'Random Forest Classifier w/ Label Encoder +␣
↪Drop '
                                               'Columns Transformer + DateTime Featurizer +␣
↪Imputer + '
                                               'One Hot Encoder + Oversampler',
                          'parameters': {...},
                          'mean_cv_score': 0.2590295567892626,
                          'standard_deviation_cv_score': 0.03951977772267373,
                          'high_variance_cv': False,
                          'training_time': 4.328300476074219,
                          'cv_data': [...],
                          'percent_better_than_baseline_all_objectives': {...},
                          'percent_better_than_baseline': 94.73650702895262,
                          'ranking_score': 0.2244398469259549,
                          'ranking_additional_objectives': {...},
                          'holdout_score': 0.2244398469259549},
                      2: {'id': 2,
                          'pipeline_name': 'Random Forest Classifier w/ Label Encoder +␣
↪Drop Columns '
```

(continues on next page)

(continued from previous page)

```
                                        'Transformer + DateTime Featurizer + Imputer +␣
↪One Hot '
                                        'Encoder + Oversampler + RF Classifier Select␣
↪From Model',
                        'pipeline_class': <class 'evalml.pipelines.binary_classification_
↪pipeline.BinaryClassificationPipeline'>,
                        'pipeline_summary': 'Random Forest Classifier w/ Label Encoder +␣
↪Drop '
                                             'Columns Transformer + DateTime Featurizer +␣
↪Imputer + '
                                             'One Hot Encoder + Oversampler + RF␣
↪Classifier Select '
                                             'From Model',
                        'parameters': {...},
                        'mean_cv_score': 0.2543819593160373,
                        'standard_deviation_cv_score': 0.045124093951055017,
                        'high_variance_cv': False,
                        'training_time': 5.085402965545654,
                        'cv_data': [...],
                        'percent_better_than_baseline_all_objectives': {...},
                        'percent_better_than_baseline': 94.83094643168255,
                        'ranking_score': 0.21914451718965428,
                        'ranking_additional_objectives': {...},
                        'holdout_score': 0.21914451718965428},
                    3: {'id': 3,
                        'pipeline_name': 'Decision Tree Classifier w/ Label Encoder +␣
↪Select '
                                        'Columns By Type Transformer + Label Encoder +␣
↪Drop '
                                        'Columns Transformer + DateTime Featurizer +␣
↪Imputer + '
                                        'Select Columns Transformer + Select Columns␣
↪Transformer + '
                                        'Label Encoder + Imputer + One Hot Encoder +␣
↪Oversampler',
                        'pipeline_class': <class 'evalml.pipelines.binary_classification_
↪pipeline.BinaryClassificationPipeline'>,
                        'pipeline_summary': 'Decision Tree Classifier w/ Label Encoder +␣
↪Select '
                                             'Columns By Type Transformer + Label Encoder␣
↪+ Drop '
                                             'Columns Transformer + DateTime Featurizer +␣
↪Imputer + '
                                             'Select Columns Transformer + Select Columns '
                                             'Transformer + Label Encoder + Imputer + One␣
↪Hot '
                                             'Encoder + Oversampler',
                        'parameters': {...},
                        'mean_cv_score': 1.4485315095019227,
                        'standard_deviation_cv_score': 0.6964965411823029,
                        'high_variance_cv': True,
                        'training_time': 3.3037452697753906,
```

(continues on next page)

(continued from previous page)

```
                        'cv_data': [...],
                        'percent_better_than_baseline_all_objectives': {...},
                        'percent_better_than_baseline': 70.56577051240947,
                        'ranking_score': 1.0028792511221052,
                        'ranking_additional_objectives': {...},
                        'holdout_score': 1.0028792511221052},
                    4: {'id': 4,
                        'pipeline_name': 'LightGBM Classifier w/ Label Encoder + Select␣
↪Columns By '
                                          'Type Transformer + Label Encoder + Drop Columns
↪'
                                          'Transformer + DateTime Featurizer + Imputer +␣
↪Select '
                                          'Columns Transformer + Select Columns␣
↪Transformer + Label '
                                          'Encoder + Imputer + One Hot Encoder +␣
↪Oversampler',
                        'pipeline_class': <class 'evalml.pipelines.binary_classification_
↪pipeline.BinaryClassificationPipeline'>,
                        'pipeline_summary': 'LightGBM Classifier w/ Label Encoder +␣
↪Select Columns '
                                             'By Type Transformer + Label Encoder + Drop␣
↪Columns '
                                             'Transformer + DateTime Featurizer + Imputer␣
↪+ Select '
                                             'Columns Transformer + Select Columns␣
↪Transformer + '
                                             'Label Encoder + Imputer + One Hot Encoder + '
                                             'Oversampler',
                        'parameters': {...},
                        'mean_cv_score': 0.2999710030621828,
                        'standard_deviation_cv_score': 0.2061756997312182,
                        'high_variance_cv': False,
                        'training_time': 3.611158609390259,
                        'cv_data': [...],
                        'percent_better_than_baseline_all_objectives': {...},
                        'percent_better_than_baseline': 93.90457488440069,
                        'ranking_score': 0.1609546813582899,
                        'ranking_additional_objectives': {...},
                        'holdout_score': 0.1609546813582899},
                    5: {'id': 5,
                        'pipeline_name': 'Extra Trees Classifier w/ Label Encoder +␣
↪Select Columns '
                                          'By Type Transformer + Label Encoder + Drop␣
↪Columns '
                                          'Transformer + DateTime Featurizer + Imputer +␣
↪Select '
                                          'Columns Transformer + Select Columns␣
↪Transformer + Label '
                                          'Encoder + Imputer + One Hot Encoder +␣
↪Oversampler',
                        'pipeline_class': <class 'evalml.pipelines.binary_classification_
↪pipeline.BinaryClassificationPipeline'>,
```

(continues on next page)

(continued from previous page)

```
                        'pipeline_summary': 'Extra Trees Classifier w/ Label Encoder +␣
↪Select '
                                            'Columns By Type Transformer + Label Encoder␣
↪+ Drop '
                                            'Columns Transformer + DateTime Featurizer +␣
↪Imputer + '
                                            'Select Columns Transformer + Select Columns '
                                            'Transformer + Label Encoder + Imputer + One␣
↪Hot '
                                            'Encoder + Oversampler',
                        'parameters': {...},
                        'mean_cv_score': 0.36134054274378125,
                        'standard_deviation_cv_score': 0.021758185101253727,
                        'high_variance_cv': False,
                        'training_time': 4.866732358932495,
                        'cv_data': [...],
                        'percent_better_than_baseline_all_objectives': {...},
                        'percent_better_than_baseline': 92.65754290567824,
                        'ranking_score': 0.3484078428021002,
                        'ranking_additional_objectives': {...},
                        'holdout_score': 0.3484078428021002},
                    6: {'id': 6,
                        'pipeline_name': 'Elastic Net Classifier w/ Label Encoder +␣
↪Select Columns '
                                         'By Type Transformer + Label Encoder + Drop␣
↪Columns '
                                         'Transformer + DateTime Featurizer + Imputer +␣
↪Standard '
                                         'Scaler + Select Columns Transformer + Select␣
↪Columns '
                                         'Transformer + Label Encoder + Imputer + One Hot␣
↪Encoder + '
                                         'Standard Scaler + Oversampler',
                        'pipeline_class': <class 'evalml.pipelines.binary_classification_
↪pipeline.BinaryClassificationPipeline'>,
                        'pipeline_summary': 'Elastic Net Classifier w/ Label Encoder +␣
↪Select '
                                            'Columns By Type Transformer + Label Encoder␣
↪+ Drop '
                                            'Columns Transformer + DateTime Featurizer +␣
↪Imputer + '
                                            'Standard Scaler + Select Columns Transformer␣
↪+ Select '
                                            'Columns Transformer + Label Encoder +␣
↪Imputer + One '
                                            'Hot Encoder + Standard Scaler + Oversampler',
                        'parameters': {...},
                        'mean_cv_score': 0.37472485974788244,
                        'standard_deviation_cv_score': 0.050026569255638,
                        'high_variance_cv': False,
                        'training_time': 4.5942182540893555,
                        'cv_data': [...],
```

(continues on next page)

(continued from previous page)

```
                        'percent_better_than_baseline_all_objectives': {...},
                        'percent_better_than_baseline': 92.38557294461829,
                        'ranking_score': 0.400375420656706,
                        'ranking_additional_objectives': {...},
                        'holdout_score': 0.400375420656706},
                    7: {'id': 7,
                        'pipeline_name': 'CatBoost Classifier w/ Label Encoder + Select␣
↪Columns By '
                                          'Type Transformer + Label Encoder + Drop Columns
↪'
                                          'Transformer + DateTime Featurizer + Imputer +␣
↪Select '
                                          'Columns Transformer + Select Columns␣
↪Transformer + Label '
                                          'Encoder + Imputer + Oversampler',
                        'pipeline_class': <class 'evalml.pipelines.binary_classification_
↪pipeline.BinaryClassificationPipeline'>,
                        'pipeline_summary': 'CatBoost Classifier w/ Label Encoder +␣
↪Select Columns '
                                             'By Type Transformer + Label Encoder + Drop␣
↪Columns '
                                             'Transformer + DateTime Featurizer + Imputer␣
↪+ Select '
                                             'Columns Transformer + Select Columns␣
↪Transformer + '
                                             'Label Encoder + Imputer + Oversampler',
                        'parameters': {...},
                        'mean_cv_score': 0.569010310977581,
                        'standard_deviation_cv_score': 0.01382528112459369,
                        'high_variance_cv': False,
                        'training_time': 2.4857382774353027,
                        'cv_data': [...],
                        'percent_better_than_baseline_all_objectives': {...},
                        'percent_better_than_baseline': 88.43768329217892,
                        'ranking_score': 0.5566549833272478,
                        'ranking_additional_objectives': {...},
                        'holdout_score': 0.5566549833272478},
                    8: {'id': 8,
                        'pipeline_name': 'XGBoost Classifier w/ Label Encoder + Select␣
↪Columns By '
                                          'Type Transformer + Label Encoder + Drop Columns
↪'
                                          'Transformer + DateTime Featurizer + Imputer +␣
↪Select '
                                          'Columns Transformer + Select Columns␣
↪Transformer + Label '
                                          'Encoder + Imputer + One Hot Encoder +␣
↪Oversampler',
                        'pipeline_class': <class 'evalml.pipelines.binary_classification_
↪pipeline.BinaryClassificationPipeline'>,
                        'pipeline_summary': 'XGBoost Classifier w/ Label Encoder + Select␣
↪Columns '
```

(continues on next page)

(continued from previous page)

```
                                            'By Type Transformer + Label Encoder + Drop␣
↪Columns '
                                            'Transformer + DateTime Featurizer + Imputer␣
↪+ Select '
                                            'Columns Transformer + Select Columns␣
↪Transformer + '
                                            'Label Encoder + Imputer + One Hot Encoder + '
                                            'Oversampler',
                        'parameters': {...},
                        'mean_cv_score': 0.2569503163235051,
                        'standard_deviation_cv_score': 0.13717967037488366,
                        'high_variance_cv': False,
                        'training_time': 4.178900480270386,
                        'cv_data': [...],
                        'percent_better_than_baseline_all_objectives': {...},
                        'percent_better_than_baseline': 94.77875729456821,
                        'ranking_score': 0.14241700777377544,
                        'ranking_additional_objectives': {...},
                        'holdout_score': 0.14241700777377544},
                    9: {'id': 9,
                        'pipeline_name': 'Logistic Regression Classifier w/ Label Encoder␣
↪+ Select '
                                          'Columns By Type Transformer + Label Encoder +␣
↪Drop '
                                          'Columns Transformer + DateTime Featurizer +␣
↪Imputer + '
                                          'Standard Scaler + Select Columns Transformer +␣
↪Select '
                                          'Columns Transformer + Label Encoder + Imputer +␣
↪One Hot '
                                          'Encoder + Standard Scaler + Oversampler',
                        'pipeline_class': <class 'evalml.pipelines.binary_classification_
↪pipeline.BinaryClassificationPipeline'>,
                        'pipeline_summary': 'Logistic Regression Classifier w/ Label␣
↪Encoder + '
                                            'Select Columns By Type Transformer + Label␣
↪Encoder + '
                                            'Drop Columns Transformer + DateTime␣
↪Featurizer + '
                                            'Imputer + Standard Scaler + Select Columns␣
↪Transformer '
                                            '+ Select Columns Transformer + Label Encoder␣
↪+ Imputer '
                                            '+ One Hot Encoder + Standard Scaler +␣
↪Oversampler',
                        'parameters': {...},
                        'mean_cv_score': 0.37436359049642465,
                        'standard_deviation_cv_score': 0.04992524856037107,
                        'high_variance_cv': False,
                        'training_time': 6.138007640838623,
                        'cv_data': [...],
                        'percent_better_than_baseline_all_objectives': {...},
```

(continues on next page)

(continued from previous page)

```
                        'percent_better_than_baseline': 92.39291395307026,
                        'ranking_score': 0.401580561387687,
                        'ranking_additional_objectives': {...},
                        'holdout_score': 0.401580561387687}},
'search_order': [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]}
```

If there are errors, such as with the Iterative Algorithm example above, we can examine these closer by accessing the `errors` field. There is one dictionary entry per pipeline fold that failed, and each entry contains the pipeline parameters with the error that was thrown and its full traceback.

[33]:
```
auto_iterative.errors
```

[33]:
```
{}
```

### 4.1.14 Parallel AutoML

By default, all pipelines in an AutoML batch are evaluated in series. Pipelines can be evaluated in parallel to improve performance during AutoML search. This is accomplished by a futures style submission and evaluation of pipelines in a batch. As of this writing, the pipelines use a threaded model for concurrent evaluation. This is similar to the currently implemented `n_jobs` parameter in the estimators, which uses increased numbers of threads to train and evaluate estimators.

#### Quick Start

To quickly use some parallelism to enhance the pipeline searching, a string can be passed through to AutoMLSearch during initialization to setup the parallel engine and client within the AutoMLSearch object. The current options are "cf_threaded", "cf_process", "dask_threaded" and "dask_process" and indicate the futures backend to use and whether to use threaded- or process-level parallelism.

[34]:
```
automl_cf_threaded = AutoMLSearch(
    X_train=X,
    y_train=y,
    problem_type="binary",
    allowed_model_families=[ModelFamily.LINEAR_MODEL],
    engine="cf_threaded",
)
automl_cf_threaded.search(interactive_plot=False)
automl_cf_threaded.close_engine()
```

```
        High coefficient of variation (cv >= 0.5) within cross validation scores.
        Decision Tree Classifier w/ Label Encoder + Select Columns By Type Transformer +␣
↪Label Encoder + Drop Columns Transformer + DateTime Featurizer + Imputer + Select␣
↪Columns Transformer + Select Columns Transformer + Label Encoder + Imputer + One Hot␣
↪Encoder + Oversampler may not perform as estimated on unseen data.
```

### Parallelism with Concurrent Futures

The `EngineBase` class is robust and extensible enough to support futures-like implementations from a variety of libraries. The `CFEngine` extends the `EngineBase` to use the native Python concurrent.futures library. The `CFEngine` supports both thread- and process-level parallelism. The type of parallelism can be chosen using either the `ThreadPoolExecutor` or the `ProcessPoolExecutor`. If either executor is passed a `max_workers` parameter, it will set the number of processes and threads spawned. If not, the default number of processes will be equal to the number of processors available and the number of threads set to five times the number of processors available.

Here, the CFEngine is invoked with default parameters, which is threaded parallelism using all available threads.

[35]:
```
from concurrent.futures import ThreadPoolExecutor

from evalml.automl.engine.cf_engine import CFEngine, CFClient

cf_engine = CFEngine(CFClient(ThreadPoolExecutor(max_workers=4)))
automl_cf_threaded = AutoMLSearch(
    X_train=X,
    y_train=y,
    problem_type="binary",
    allowed_model_families=[ModelFamily.LINEAR_MODEL],
    engine=cf_engine,
)
automl_cf_threaded.search(interactive_plot=False)
automl_cf_threaded.close_engine()
```

```
        High coefficient of variation (cv >= 0.5) within cross validation scores.
        Decision Tree Classifier w/ Label Encoder + Select Columns By Type Transformer +␣
↪Label Encoder + Drop Columns Transformer + DateTime Featurizer + Imputer + Select␣
↪Columns Transformer + Select Columns Transformer + Label Encoder + Imputer + One Hot␣
↪Encoder + Oversampler may not perform as estimated on unseen data.
```

Note: the cell demonstrating process-level parallelism is a markdown due to incompatibility with our ReadTheDocs build. It can be run successfully locally.

```
from concurrent.futures import ProcessPoolExecutor

# Repeat the process but using process-level parallelism\
cf_engine = CFEngine(CFClient(ProcessPoolExecutor(max_workers=2)))
automl_cf_process = AutoMLSearch(X_train=X, y_train=y,
                                 problem_type="binary",
                                 engine="cf_process")
automl_cf_process.search(interactive_plot = False)
automl_cf_process.close_engine()
```

### Parallelism with Dask

Thread or process level parallelism can be explicitly invoked for the `DaskEngine` (as well as the `CFEngine`). The `processes` can be set to `True` and the number of processes set using `n_workers`. If `processes` is set to `False`, then the resulting parallelism will be threaded and `n_workers` will represent the threads used. Examples of both follow.

[36]:
```
from dask.distributed import LocalCluster

from evalml.automl.engine import DaskEngine

dask_engine_p2 = DaskEngine(cluster=LocalCluster(processes=True, n_workers=2))
automl_dask_p2 = AutoMLSearch(
    X_train=X,
    y_train=y,
    problem_type="binary",
    allowed_model_families=[ModelFamily.LINEAR_MODEL],
    engine=dask_engine_p2,
)
automl_dask_p2.search(interactive_plot=False)

# Explicitly shutdown the automl object's LocalCluster
automl_dask_p2.close_engine()
```

```
        High coefficient of variation (cv >= 0.5) within cross validation scores.
        Decision Tree Classifier w/ Label Encoder + Select Columns By Type Transformer +␣
↪Label Encoder + Drop Columns Transformer + DateTime Featurizer + Imputer + Select␣
↪Columns Transformer + Select Columns Transformer + Label Encoder + Imputer + One Hot␣
↪Encoder + Oversampler may not perform as estimated on unseen data.
Inside a Dask worker with daemon=True, setting n_jobs=1.
Possible work-arounds:
- dask.config.set({'distributed.worker.daemon': False})
- set the environment variable DASK_DISTRIBUTED__WORKER__DAEMON=False
before creating your Dask cluster.
Inside a Dask worker with daemon=True, setting n_jobs=1.
Possible work-arounds:
- dask.config.set({'distributed.worker.daemon': False})
- set the environment variable DASK_DISTRIBUTED__WORKER__DAEMON=False
before creating your Dask cluster.
Inside a Dask worker with daemon=True, setting n_jobs=1.
Possible work-arounds:
- dask.config.set({'distributed.worker.daemon': False})
- set the environment variable DASK_DISTRIBUTED__WORKER__DAEMON=False
before creating your Dask cluster.
Inside a Dask worker with daemon=True, setting n_jobs=1.
Possible work-arounds:
- dask.config.set({'distributed.worker.daemon': False})
- set the environment variable DASK_DISTRIBUTED__WORKER__DAEMON=False
before creating your Dask cluster.
Inside a Dask worker with daemon=True, setting n_jobs=1.
Possible work-arounds:
- dask.config.set({'distributed.worker.daemon': False})
- set the environment variable DASK_DISTRIBUTED__WORKER__DAEMON=False
before creating your Dask cluster.
Inside a Dask worker with daemon=True, setting n_jobs=1.
```

(continues on next page)

(continued from previous page)

```
Possible work-arounds:
- dask.config.set({'distributed.worker.daemon': False})
- set the environment variable DASK_DISTRIBUTED__WORKER__DAEMON=False
before creating your Dask cluster.
Inside a Dask worker with daemon=True, setting n_jobs=1.
Possible work-arounds:
- dask.config.set({'distributed.worker.daemon': False})
- set the environment variable DASK_DISTRIBUTED__WORKER__DAEMON=False
before creating your Dask cluster.
Inside a Dask worker with daemon=True, setting n_jobs=1.
Possible work-arounds:
- dask.config.set({'distributed.worker.daemon': False})
- set the environment variable DASK_DISTRIBUTED__WORKER__DAEMON=False
before creating your Dask cluster.
Inside a Dask worker with daemon=True, setting n_jobs=1.
Possible work-arounds:
- dask.config.set({'distributed.worker.daemon': False})
- set the environment variable DASK_DISTRIBUTED__WORKER__DAEMON=False
before creating your Dask cluster.
Inside a Dask worker with daemon=True, setting n_jobs=1.
Possible work-arounds:
- dask.config.set({'distributed.worker.daemon': False})
- set the environment variable DASK_DISTRIBUTED__WORKER__DAEMON=False
before creating your Dask cluster.
Inside a Dask worker with daemon=True, setting n_jobs=1.
Possible work-arounds:
- dask.config.set({'distributed.worker.daemon': False})
- set the environment variable DASK_DISTRIBUTED__WORKER__DAEMON=False
before creating your Dask cluster.
Inside a Dask worker with daemon=True, setting n_jobs=1.
Possible work-arounds:
- dask.config.set({'distributed.worker.daemon': False})
- set the environment variable DASK_DISTRIBUTED__WORKER__DAEMON=False
before creating your Dask cluster.
Inside a Dask worker with daemon=True, setting n_jobs=1.
Possible work-arounds:
- dask.config.set({'distributed.worker.daemon': False})
- set the environment variable DASK_DISTRIBUTED__WORKER__DAEMON=False
before creating your Dask cluster.
Inside a Dask worker with daemon=True, setting n_jobs=1.
Possible work-arounds:
- dask.config.set({'distributed.worker.daemon': False})
- set the environment variable DASK_DISTRIBUTED__WORKER__DAEMON=False
before creating your Dask cluster.
Inside a Dask worker with daemon=True, setting n_jobs=1.
Possible work-arounds:
- dask.config.set({'distributed.worker.daemon': False})
- set the environment variable DASK_DISTRIBUTED__WORKER__DAEMON=False
before creating your Dask cluster.
Inside a Dask worker with daemon=True, setting n_jobs=1.
Possible work-arounds:
- dask.config.set({'distributed.worker.daemon': False})
```

(continues on next page)

(continued from previous page)

```
- set the environment variable DASK_DISTRIBUTED__WORKER__DAEMON=False
before creating your Dask cluster.
Inside a Dask worker with daemon=True, setting n_jobs=1.
Possible work-arounds:
- dask.config.set({'distributed.worker.daemon': False})
- set the environment variable DASK_DISTRIBUTED__WORKER__DAEMON=False
before creating your Dask cluster.
Inside a Dask worker with daemon=True, setting n_jobs=1.
Possible work-arounds:
- dask.config.set({'distributed.worker.daemon': False})
- set the environment variable DASK_DISTRIBUTED__WORKER__DAEMON=False
before creating your Dask cluster.
Inside a Dask worker with daemon=True, setting n_jobs=1.
Possible work-arounds:
- dask.config.set({'distributed.worker.daemon': False})
- set the environment variable DASK_DISTRIBUTED__WORKER__DAEMON=False
before creating your Dask cluster.
Inside a Dask worker with daemon=True, setting n_jobs=1.
Possible work-arounds:
- dask.config.set({'distributed.worker.daemon': False})
- set the environment variable DASK_DISTRIBUTED__WORKER__DAEMON=False
before creating your Dask cluster.
Inside a Dask worker with daemon=True, setting n_jobs=1.
Possible work-arounds:
- dask.config.set({'distributed.worker.daemon': False})
- set the environment variable DASK_DISTRIBUTED__WORKER__DAEMON=False
before creating your Dask cluster.
Inside a Dask worker with daemon=True, setting n_jobs=1.
Possible work-arounds:
- dask.config.set({'distributed.worker.daemon': False})
- set the environment variable DASK_DISTRIBUTED__WORKER__DAEMON=False
before creating your Dask cluster.
Inside a Dask worker with daemon=True, setting n_jobs=1.
Possible work-arounds:
- dask.config.set({'distributed.worker.daemon': False})
- set the environment variable DASK_DISTRIBUTED__WORKER__DAEMON=False
before creating your Dask cluster.
Inside a Dask worker with daemon=True, setting n_jobs=1.
Possible work-arounds:
- dask.config.set({'distributed.worker.daemon': False})
- set the environment variable DASK_DISTRIBUTED__WORKER__DAEMON=False
before creating your Dask cluster.
```

[37]:

```
dask_engine_t4 = DaskEngine(cluster=LocalCluster(processes=False, n_workers=4))

automl_dask_t4 = AutoMLSearch(
    X_train=X,
    y_train=y,
    problem_type="binary",
    allowed_model_families=[ModelFamily.LINEAR_MODEL],
    engine=dask_engine_t4,
)
```

(continues on next page)

(continued from previous page)

```
automl_dask_t4.search(interactive_plot=False)
automl_dask_t4.close_engine()
```

```
        High coefficient of variation (cv >= 0.5) within cross validation scores.
        Decision Tree Classifier w/ Label Encoder + Select Columns By Type Transformer +␣
↪Label Encoder + Drop Columns Transformer + DateTime Featurizer + Imputer + Select␣
↪Columns Transformer + Select Columns Transformer + Label Encoder + Imputer + One Hot␣
↪Encoder + Oversampler may not perform as estimated on unseen data.
```

As we can see, a significant performance gain can result from simply using something other than the default `SequentialEngine`, ranging from a 100% speed up with multiple processes to 500% speedup with multiple threads!

[38]:
```
print("Sequential search duration: %s" % str(automl.search_duration))
print(
    "Concurrent futures (threaded) search duration: %s"
    % str(automl_cf_threaded.search_duration)
)
print("Dask (two processes) search duration: %s" % str(automl_dask_p2.search_duration))
print("Dask (four threads)search duration: %s" % str(automl_dask_t4.search_duration))
```

```
Sequential search duration: 40.80535435676575
Concurrent futures (threaded) search duration: 32.44133520126343
Dask (two processes) search duration: 34.83338689804077
Dask (four threads)search duration: 37.224250078201294
```

## 4.2 Pipelines

EvalML pipelines represent a sequence of operations to be applied to data, where each operation is either a data transformation or an ML modeling algorithm.

A pipeline holds a combination of one or more components, which will be applied to new input data in sequence.

Each component and pipeline supports a set of parameters which configure its behavior. The AutoML search process seeks to find the combination of pipeline structure and pipeline parameters which perform the best on the data.

### 4.2.1 Defining a Pipeline Instance

Pipeline instances can be instantiated using any of the following classes:

- `RegressionPipeline`
- `BinaryClassificationPipeline`
- `MulticlassClassificationPipeline`
- `TimeSeriesRegressionPipeline`
- `TimeSeriesBinaryClassificationPipeline`
- `TimeSeriesMulticlassClassificationPipeline`

The class you want to use will depend on your problem type. The only required parameter input for instantiating a pipeline instance is `component_graph`, which can be a `ComponentGraph` instance, a list, or a dictionary containing a sequence of components to be fit and evaluated.

A `component_graph` list is the default representation, which represents a linear order of transforming components with an estimator as the final component. A `component_graph` dictionary is used to represent a non-linear graph of components, where the key is a unique name for each component and the value is a list with the component's class as the first element and any parents of the component as the following element(s). For these two `component_graph` formats, each component can be provided as a reference to the component class for custom components, and as either a string name or as a reference to the component class for components defined in EvalML.

If you choose to provide a `ComponentGraph` instance and want to set custom parameters for your pipeline, set it through the pipeline initialization rather than `ComponentGraph.instantiate()`.

[1]:
```
from evalml.pipelines import MulticlassClassificationPipeline, ComponentGraph

component_graph_as_list = ["Imputer", "Random Forest Classifier"]
MulticlassClassificationPipeline(component_graph=component_graph_as_list)
```

[1]:
```
pipeline = MulticlassClassificationPipeline(component_graph={'Imputer': ['Imputer', 'X',
↪'y'], 'Random Forest Classifier': ['Random Forest Classifier', 'Imputer.x', 'y']},␣
↪parameters={'Imputer':{'categorical_impute_strategy': 'most_frequent', 'numeric_impute_
↪strategy': 'mean', 'boolean_impute_strategy': 'most_frequent', 'categorical_fill_value
↪': None, 'numeric_fill_value': None, 'boolean_fill_value': None}, 'Random Forest␣
↪Classifier':{'n_estimators': 100, 'max_depth': 6, 'n_jobs': -1}}, random_seed=0)
```

[2]:
```
component_graph_as_dict = {
    "Imputer": ["Imputer", "X", "y"],
    "Encoder": ["One Hot Encoder", "Imputer.x", "y"],
    "Random Forest Clf": ["Random Forest Classifier", "Encoder.x", "y"],
    "Elastic Net Clf": ["Elastic Net Classifier", "Encoder.x", "y"],
    "Final Estimator": [
        "Logistic Regression Classifier",
        "Random Forest Clf.x",
        "Elastic Net Clf.x",
        "y",
    ],
}

MulticlassClassificationPipeline(component_graph=component_graph_as_dict)
```

[2]:
```
pipeline = MulticlassClassificationPipeline(component_graph={'Imputer': ['Imputer', 'X',
↪'y'], 'Encoder': ['One Hot Encoder', 'Imputer.x', 'y'], 'Random Forest Clf': ['Random␣
↪Forest Classifier', 'Encoder.x', 'y'], 'Elastic Net Clf': ['Elastic Net Classifier',
↪'Encoder.x', 'y'], 'Final Estimator': ['Logistic Regression Classifier', 'Random␣
↪Forest Clf.x', 'Elastic Net Clf.x', 'y']}, parameters={'Imputer':{'categorical_impute_
↪strategy': 'most_frequent', 'numeric_impute_strategy': 'mean', 'boolean_impute_strategy
↪': 'most_frequent', 'categorical_fill_value': None, 'numeric_fill_value': None,
↪'boolean_fill_value': None}, 'Encoder':{'top_n': 10, 'features_to_encode': None,
↪'categories': None, 'drop': 'if_binary', 'handle_unknown': 'ignore', 'handle_missing':
↪'error'}, 'Random Forest Clf':{'n_estimators': 100, 'max_depth': 6, 'n_jobs': -1},
↪'Elastic Net Clf':{'penalty': 'elasticnet', 'C': 1.0, 'l1_ratio': 0.15, 'n_jobs': -1,
↪'multi_class': 'auto', 'solver': 'saga'}, 'Final Estimator':{'penalty': 'l2', 'C': 1.0,
↪ 'n_jobs': -1, 'multi_class': 'auto', 'solver': 'lbfgs'}}, random_seed=0)
```

[3]:
```
cg = ComponentGraph(component_graph_as_dict)

# set parameters in the pipeline rather than through cg.instantiate()
MulticlassClassificationPipeline(component_graph=cg, parameters={})
```

[3]:
```
pipeline = MulticlassClassificationPipeline(component_graph={'Imputer': ['Imputer', 'X',
↪'y'], 'Encoder': ['One Hot Encoder', 'Imputer.x', 'y'], 'Random Forest Clf': ['Random␣
↪Forest Classifier', 'Encoder.x', 'y'], 'Elastic Net Clf': ['Elastic Net Classifier',
↪'Encoder.x', 'y'], 'Final Estimator': ['Logistic Regression Classifier', 'Random␣
↪Forest Clf.x', 'Elastic Net Clf.x', 'y']}, parameters={'Imputer':{'categorical_impute_
↪strategy': 'most_frequent', 'numeric_impute_strategy': 'mean', 'boolean_impute_strategy
↪': 'most_frequent', 'categorical_fill_value': None, 'numeric_fill_value': None,
↪'boolean_fill_value': None}, 'Encoder':{'top_n': 10, 'features_to_encode': None,
↪'categories': None, 'drop': 'if_binary', 'handle_unknown': 'ignore', 'handle_missing':
↪'error'}, 'Random Forest Clf':{'n_estimators': 100, 'max_depth': 6, 'n_jobs': -1},
↪'Elastic Net Clf':{'penalty': 'elasticnet', 'C': 1.0, 'l1_ratio': 0.15, 'n_jobs': -1,
↪'multi_class': 'auto', 'solver': 'saga'}, 'Final Estimator':{'penalty': 'l2', 'C': 1.0,
↪ 'n_jobs': -1, 'multi_class': 'auto', 'solver': 'lbfgs'}}, random_seed=0)
```

If you're using your own *custom components* you can refer to them like so:

[4]:
```
from evalml.pipelines.components import Transformer


class NewTransformer(Transformer):
    name = "New Transformer"
    hyperparameter_ranges = {"parameter_1": ["a", "b", "c"]}

    def __init__(self, parameter_1=1, random_seed=0):
        parameters = {"parameter_1": parameter_1}
        super().__init__(parameters=parameters, random_seed=random_seed)

    def transform(self, X, y=None):
        # Your code here!
        return X


MulticlassClassificationPipeline([NewTransformer, "Random Forest Classifier"])
```

[4]:
```
pipeline = MulticlassClassificationPipeline(component_graph={'New Transformer':␣
↪[NewTransformer, 'X', 'y'], 'Random Forest Classifier': ['Random Forest Classifier',
↪'New Transformer.x', 'y']}, parameters={'New Transformer':{'parameter_1': 1}, 'Random␣
↪Forest Classifier':{'n_estimators': 100, 'max_depth': 6, 'n_jobs': -1}}, random_seed=0)
```

### 4.2.2 Pipeline Usage

All pipelines define the following methods:

- `fit` fits each component on the provided training data, in order.
- `predict` computes the predictions of the component graph on the provided data.
- `score` computes the value of *an objective* on the provided data.

[5]:
```
from evalml.demos import load_wine

X, y = load_wine()

pipeline = MulticlassClassificationPipeline(
```

(continues on next page)

(continued from previous page)

```
    component_graph={
        "Label Encoder": ["Label Encoder", "X", "y"],
        "Imputer": ["Imputer", "X", "Label Encoder.y"],
        "Random Forest Classifier": [
            "Random Forest Classifier",
            "Imputer.x",
            "Label Encoder.y",
        ],
    }
)
pipeline.fit(X, y)
print(pipeline.predict(X))
print(pipeline.score(X, y, objectives=["log loss multiclass"]))
```

```
         Number of Features
Numeric                  13

Number of training examples: 178
Targets
class_1    39.89%
class_0    33.15%
class_2    26.97%
Name: target, dtype: object
0      class_0
1      class_0
2      class_0
3      class_0
4      class_0
        ...
173    class_2
174    class_2
175    class_2
176    class_2
177    class_2
Length: 178, dtype: category
Categories (3, object): ['class_0', 'class_1', 'class_2']
OrderedDict([('Log Loss Multiclass', 0.04132737017536072)])
```

### 4.2.3 Custom Name

By default, a pipeline's name is created using the component graph that makes up the pipeline. E.g. A pipeline with an imputer, one-hot encoder, and logistic regression classifier will have the name 'Logistic Regression Classifier w/ Imputer + One Hot Encoder'.

If you'd like to override the pipeline's name attribute, you can set the `custom_name` parameter when initalizing a pipeline, like so:

[6]:
```
component_graph = ["Imputer", "One Hot Encoder", "Logistic Regression Classifier"]
pipeline = MulticlassClassificationPipeline(component_graph)
print("Pipeline with default name:", pipeline.name)
```

(continues on next page)

(continued from previous page)

```
pipeline_with_name = MulticlassClassificationPipeline(
    component_graph, custom_name="My cool custom pipeline"
)
print("Pipeline with custom name:", pipeline_with_name.name)
```

```
Pipeline with default name: Logistic Regression Classifier w/ Imputer + One Hot Encoder
Pipeline with custom name: My cool custom pipeline
```

### 4.2.4 Pipeline Parameters

You can also pass in custom parameters by using the `parameters` parameter, which will then be used when instantiating each component in `component_graph`. The parameters dictionary needs to be in the format of a two-layered dictionary where the key-value pairs are the component name and corresponding component parameters dictionary. The component parameters dictionary consists of (parameter name, parameter values) key-value pairs.

An example will be shown below. The API reference for component parameters can also be found *here*.

[7]:
```
parameters = {
    "Imputer": {
        "categorical_impute_strategy": "most_frequent",
        "numeric_impute_strategy": "median",
    },
    "Logistic Regression Classifier": {
        "penalty": "l2",
        "C": 1.0,
    },
}
component_graph = [
    "Imputer",
    "One Hot Encoder",
    "Standard Scaler",
    "Logistic Regression Classifier",
]
MulticlassClassificationPipeline(component_graph=component_graph, parameters=parameters)
```

[7]:
```
pipeline = MulticlassClassificationPipeline(component_graph={'Imputer': ['Imputer', 'X',
↪'y'], 'One Hot Encoder': ['One Hot Encoder', 'Imputer.x', 'y'], 'Standard Scaler': [
↪'Standard Scaler', 'One Hot Encoder.x', 'y'], 'Logistic Regression Classifier': [
↪'Logistic Regression Classifier', 'Standard Scaler.x', 'y']}, parameters={'Imputer':{
↪'categorical_impute_strategy': 'most_frequent', 'numeric_impute_strategy': 'median',
↪'boolean_impute_strategy': 'most_frequent', 'categorical_fill_value': None, 'numeric_
↪fill_value': None, 'boolean_fill_value': None}, 'One Hot Encoder':{'top_n': 10,
↪'features_to_encode': None, 'categories': None, 'drop': 'if_binary', 'handle_unknown':
↪'ignore', 'handle_missing': 'error'}, 'Logistic Regression Classifier':{'penalty': 'l2
↪', 'C': 1.0, 'n_jobs': -1, 'multi_class': 'auto', 'solver': 'lbfgs'}}, random_seed=0)
```

### 4.2.5 Pipeline Description

You can call `.graph()` to see each component and its parameters. Each component takes in data and feeds it to the next.

[8]:
```
component_graph = [
    "Imputer",
    "One Hot Encoder",
    "Standard Scaler",
    "Logistic Regression Classifier",
]
pipeline = MulticlassClassificationPipeline(
    component_graph=component_graph, parameters=parameters
)
pipeline.graph()
```

[8]:

[9]:
```
component_graph_as_dict = {
    "Imputer": ["Imputer", "X", "y"],
    "Encoder": ["One Hot Encoder", "Imputer.x", "y"],
    "Random Forest Clf": ["Random Forest Classifier", "Encoder.x", "y"],
    "Elastic Net Clf": ["Elastic Net Classifier", "Encoder.x", "y"],
    "Final Estimator": [
        "Logistic Regression Classifier",
        "Random Forest Clf.x",
        "Elastic Net Clf.x",
        "y",
    ],
}

nonlinear_pipeline = MulticlassClassificationPipeline(
    component_graph=component_graph_as_dict
)
nonlinear_pipeline.graph()
```

[9]:

You can see a textual representation of the pipeline by calling `.describe()`:

[10]:
```
pipeline.describe()
```

```
*******************************************************************************
* Logistic Regression Classifier w/ Imputer + One Hot Encoder + Standard Scaler *
*******************************************************************************

Problem Type: multiclass
Model Family: Linear

Pipeline Steps
==============
1. Imputer
         * categorical_impute_strategy : most_frequent
         * numeric_impute_strategy : median
         * boolean_impute_strategy : most_frequent
         * categorical_fill_value : None
```

(continues on next page)

(continued from previous page)

```
        * numeric_fill_value : None
        * boolean_fill_value : None
2. One Hot Encoder
        * top_n : 10
        * features_to_encode : None
        * categories : None
        * drop : if_binary
        * handle_unknown : ignore
        * handle_missing : error
3. Standard Scaler
4. Logistic Regression Classifier
        * penalty : l2
        * C : 1.0
        * n_jobs : -1
        * multi_class : auto
        * solver : lbfgs
```

[11]:
```
nonlinear_pipeline.describe()
```

```
*****************************************************************************************************************
* Logistic Regression Classifier w/ Imputer + One Hot Encoder + Random Forest Classifier␣
↪+ Elastic Net Classifier *
*****************************************************************************************************************

Problem Type: multiclass
Model Family: Linear

Pipeline Steps
==============
1. Imputer
        * categorical_impute_strategy : most_frequent
        * numeric_impute_strategy : mean
        * boolean_impute_strategy : most_frequent
        * categorical_fill_value : None
        * numeric_fill_value : None
        * boolean_fill_value : None
2. One Hot Encoder
        * top_n : 10
        * features_to_encode : None
        * categories : None
        * drop : if_binary
        * handle_unknown : ignore
        * handle_missing : error
3. Random Forest Classifier
        * n_estimators : 100
        * max_depth : 6
        * n_jobs : -1
4. Elastic Net Classifier
        * penalty : elasticnet
        * C : 1.0
        * l1_ratio : 0.15
```

(continues on next page)

(continued from previous page)

```
        * n_jobs : -1
        * multi_class : auto
        * solver : saga
5. Logistic Regression Classifier
        * penalty : l2
        * C : 1.0
        * n_jobs : -1
        * multi_class : auto
        * solver : lbfgs
```

### 4.2.6 Component Graph

You can use `pipeline.get_component(name)` and provide the component name to access any component (API reference *here*):

[12]:
```
pipeline.get_component("Imputer")
```

[12]:
```
Imputer(categorical_impute_strategy='most_frequent', numeric_impute_strategy='median',␣
↪boolean_impute_strategy='most_frequent', categorical_fill_value=None, numeric_fill_
↪value=None, boolean_fill_value=None)
```

[13]:
```
nonlinear_pipeline.get_component("Elastic Net Clf")
```

[13]:
```
ElasticNetClassifier(penalty='elasticnet', C=1.0, l1_ratio=0.15, n_jobs=-1, multi_class=
↪'auto', solver='saga')
```

Alternatively, you can index directly into the pipeline to get a component

[14]:
```
first_component = pipeline[0]
print(first_component.name)
```

```
Imputer
```

[15]:
```
nonlinear_pipeline["Final Estimator"]
```

[15]:
```
LogisticRegressionClassifier(penalty='l2', C=1.0, n_jobs=-1, multi_class='auto', solver=
↪'lbfgs')
```

### 4.2.7 Pipeline Estimator

EvalML enforces that the last component of a linear pipeline is an estimator. You can access this estimator directly by using `pipeline.estimator`.

[16]:
```
pipeline.estimator
```

[16]:
```
LogisticRegressionClassifier(penalty='l2', C=1.0, n_jobs=-1, multi_class='auto', solver=
↪'lbfgs')
```

### 4.2.8 Input Feature Names

After a pipeline is fitted, you can access a pipeline's `input_feature_names` attribute to obtain a dictionary containing a list of feature names passed to each component of the pipeline. This could be especially useful for debugging where a feature might have been dropped or detecting unexpected behavior.

[17]:
```
pipeline = MulticlassClassificationPipeline(["Imputer", "Random Forest Classifier"])
pipeline.fit(X, y)
pipeline.input_feature_names
```

[17]:
```
{'Imputer': ['alcohol',
  'malic_acid',
  'ash',
  'alcalinity_of_ash',
  'magnesium',
  'total_phenols',
  'flavanoids',
  'nonflavanoid_phenols',
  'proanthocyanins',
  'color_intensity',
  'hue',
  'od280/od315_of_diluted_wines',
  'proline'],
 'Random Forest Classifier': ['alcohol',
  'malic_acid',
  'ash',
  'alcalinity_of_ash',
  'magnesium',
  'total_phenols',
  'flavanoids',
  'nonflavanoid_phenols',
  'proanthocyanins',
  'color_intensity',
  'hue',
  'od280/od315_of_diluted_wines',
  'proline']}
```

### 4.2.9 Binary Classification Pipeline Thresholds

For binary classification pipelines, you can choose to tune the decision boundary threshold, which allows the pipeline to distinguish predictions from positive to negative. The default boundary, if none is set, is 0.5, which means that predictions with a probability of >= `0.5` are classified as the positive class, while all others are negative.

You can use the binary classification pipeline's `optimize_thresholds` method to choose the best threshold for an objective, or it can be manually set. EvalML's *AutoMLSearch* uses `optimize_thresholds` by default for `binary` problems, and it uses `F1` as the default objective to optimize on. This can be turned off by passing in `optimize_thresholds=False`, or you can changed the objective used by changing the `objective` or `alternate_thresholding_objective` arguments.

[18]:
```
from evalml.demos import load_breast_cancer
from evalml.pipelines import BinaryClassificationPipeline

X, y = load_breast_cancer()
```

(continues on next page)

(continued from previous page)

```
X_to_predict = X.tail(10)

bcp = BinaryClassificationPipeline(
    {
        "Imputer": ["Imputer", "X", "y"],
        "Label Encoder": ["Label Encoder", "Imputer.x", "y"],
        "RFC": ["Random Forest Classifier", "Imputer.x", "Label Encoder.y"],
    }
)
bcp.fit(X, y)

predict_proba = bcp.predict_proba(X_to_predict)
predict_proba
```

```
         Number of Features
Numeric                  30

Number of training examples: 569
Targets
benign       62.74%
malignant    37.26%
Name: target, dtype: object
```

[18]:

```
       benign  malignant
559  0.925711   0.074289
560  0.939512   0.060488
561  0.991177   0.008823
562  0.010155   0.989845
563  0.000155   0.999845
564  0.000100   0.999900
565  0.000155   0.999845
566  0.011528   0.988472
567  0.000155   0.999845
568  0.994452   0.005548
```

[19]:

```
# view the current threshold
print("The threshold is {}".format(bcp.threshold))

# view the first few predictions
print(bcp.predict(X_to_predict))
```

```
The threshold is None
559       benign
560       benign
561       benign
562    malignant
563    malignant
564    malignant
565    malignant
566    malignant
567    malignant
568       benign
dtype: category
```

(continues on next page)

(continued from previous page)

```
Categories (2, object): ['benign', 'malignant']
```

Note that the default threshold above is `None`, which means that the pipeline defaults to using 0.5 as the threshold.

You can manually set the threshold as well:

[20]:
```
# you can manually set the threshold
bcp.threshold = 0.99
# view the threshold
print("The threshold is {}".format(bcp.threshold))

# view the first few predictions
print(bcp.predict(X_to_predict))
```

```
The threshold is 0.99
559       benign
560       benign
561       benign
562       benign
563    malignant
564    malignant
565    malignant
566       benign
567    malignant
568       benign
Name: malignant, dtype: category
Categories (2, object): ['benign', 'malignant']
```

However, the best way to set the threshold is by using the pipeline's `optimize_threshold` method. This takes in the predicted values, as well as the true values and objective to optimize with, and it finds the best threshold to maximize this objective value.

This method is best used with validation data, since optimizing on training data could lead to overfitting and optimizing on test data would introduce large biases.

Below walks through threshold tuning using the `F1` objective.

[21]:
```
from evalml.objectives import F1

# get predictions for positive class only
predict_proba = predict_proba.iloc[:, -1]
bcp.optimize_threshold(X_to_predict, y.tail(10), predict_proba, F1())

print("The new threshold is {}".format(bcp.threshold))

# view the first few predictions
print(bcp.predict(X_to_predict))
```

```
The new threshold is 0.13521817340545206
559       benign
560       benign
561       benign
562    malignant
563    malignant
564    malignant
```

(continues on next page)

(continued from previous page)

```
565    malignant
566    malignant
567    malignant
568       benign
Name: malignant, dtype: category
Categories (2, object): ['benign', 'malignant']
```

### 4.2.10 Grabbing rows near the decision boundary

For binary classification problems, you can also look at the rows closest to the decision boundary by using `rows_of_interest`. This method returns the indices of interest, which can then be used to obtain the subset of the data that falls closest to the decision boundary. This can help with further analysis of the model, and can give you better understanding of what rows the model could be having trouble with.

`rows_of_interest` takes in an `epsilon` parameter (defaulted to `0.1`), which determines which rows to return. The rows that are returned are the rows where the probability of it being in the positive class fall between the `threshold +- epsilon` range. Increase the `epsilon` value to get more rows, and decrease it to get fewer rows.

Below is a walkthrough of using `rows_of_interest`, building off the previous pipeline which is already thresholded.

[22]:
```
from evalml.pipelines.utils import rows_of_interest

indices = rows_of_interest(bcp, X, y, types="all")
X.iloc[indices].head()
```

[22]:
```
     mean radius  mean texture  mean perimeter  mean area  mean smoothness  \
375        16.17         16.07          106.30      788.5          0.09880
472        14.92         14.93           96.45      686.9          0.08098
191        12.77         21.41           82.02      507.4          0.08749
290        14.41         19.73           96.03      651.0          0.08757
413        14.99         22.11           97.53      693.7          0.08515

     mean compactness  mean concavity  mean concave points  mean symmetry  \
375           0.14380         0.06651              0.05397         0.1990
472           0.08549         0.05539              0.03221         0.1687
191           0.06601         0.03112              0.02864         0.1694
290           0.16760         0.13620              0.06602         0.1714
413           0.10250         0.06859              0.03876         0.1944

     mean fractal dimension  ...  worst radius  worst texture  \
375                 0.06572  ...         16.97          19.14
472                 0.05669  ...         17.18          18.22
191                 0.06287  ...         13.75          23.50
290                 0.07192  ...         15.77          22.13
413                 0.05913  ...         16.76          31.55

     worst perimeter  worst area  worst smoothness  worst compactness  \
375           113.10       861.5           0.12350            0.25500
472           112.00       906.6           0.10650            0.27910
191            89.04       579.5           0.09388            0.08978
290           101.70       767.3           0.09983            0.24720
413           110.20       867.1           0.10770            0.33450
```

(continues on next page)

(continued from previous page)

```
     worst concavity  worst concave points  worst symmetry  \
375          0.21140               0.12510          0.3153
472          0.31510               0.11470          0.2688
191          0.05186               0.04773          0.2179
290          0.22200               0.10210          0.2272
413          0.31140               0.13080          0.3163

     worst fractal dimension
375                  0.08960
472                  0.08273
191                  0.06871
290                  0.08799
413                  0.09251

[5 rows x 30 columns]
```

You can see what the probabilities are for these rows to determine how close they are to the new pipeline threshold. X is used here for brevity.

[23]:
```
pred_proba = bcp.predict_proba(X)
pos_value_proba = pred_proba.iloc[:, -1]
pos_value_proba.iloc[indices].head()
```

[23]:
```
375    0.133328
472    0.130808
191    0.128998
290    0.127939
413    0.149718
Name: malignant, dtype: float64
```

### 4.2.11 Saving and Loading Pipelines

You can save and load trained or untrained pipeline instances using the Python pickle format, like so:

[24]:
```
import pickle

pipeline_to_pickle = BinaryClassificationPipeline(
    ["Imputer", "Random Forest Classifier"]
)

with open("pipeline.pkl", "wb") as f:
    pickle.dump(pipeline_to_pickle, f)

pickled_pipeline = None
with open("pipeline.pkl", "rb") as f:
    pickled_pipeline = pickle.load(f)

assert pickled_pipeline == pipeline_to_pickle
pickled_pipeline.fit(X, y)
```

[24]:
```
pipeline = BinaryClassificationPipeline(component_graph={'Imputer': ['Imputer', 'X', 'y
↪'], 'Random Forest Classifier': ['Random Forest Classifier', 'Imputer.x', 'y']},␣
↪parameters={'Imputer':{'categorical_impute_strategy': 'most_frequent', 'numeric_impute_
↪strategy': 'mean', 'boolean_impute_strategy': 'most_frequent', 'categorical_fill_value
↪': None, 'numeric_fill_value': None, 'boolean_fill_value': None}, 'Random Forest␣
↪Classifier':{'n_estimators': 100, 'max_depth': 6, 'n_jobs': -1}}, random_seed=0)
```

(continues on next page)

(continued from previous page)

### 4.2.12 Generate Code

Once you have instantiated a pipeline, you can generate string Python code to recreate this pipeline, which can then be saved and run elsewhere with EvalML. `generate_pipeline_code` requires a pipeline instance as the input. It can also handle custom components, but it won't return the code required to define the component. Note that any external libraries used in creating the pipeline instance will also need to be imported to execute the returned code.

Code generation is not yet supported for nonlinear pipelines.

[25]:
```
from evalml.pipelines.utils import generate_pipeline_code
from evalml.pipelines import BinaryClassificationPipeline
import pandas as pd
from evalml.utils import infer_feature_types
from skopt.space import Integer


class MyDropNullColumns(Transformer):
    """Transformer to drop features whose percentage of NaN values exceeds a specified␣
↪threshold"""

    name = "My Drop Null Columns Transformer"
    hyperparameter_ranges = {}

    def __init__(self, pct_null_threshold=1.0, random_seed=0, **kwargs):
        """Initalizes an transformer to drop features whose percentage of NaN values␣
↪exceeds a specified threshold.

        Args:
            pct_null_threshold(float): The percentage of NaN values in an input feature␣
↪to drop.
                Must be a value between [0, 1] inclusive. If equal to 0.0, will drop␣
↪columns with any null values.
                If equal to 1.0, will drop columns with all null values. Defaults to 0.
↪95.
        """
        if pct_null_threshold < 0 or pct_null_threshold > 1:
            raise ValueError(
                "pct_null_threshold must be a float between 0 and 1, inclusive."
            )
        parameters = {"pct_null_threshold": pct_null_threshold}
        parameters.update(kwargs)

        self._cols_to_drop = None
        super().__init__(
            parameters=parameters, component_obj=None, random_seed=random_seed
        )

    def fit(self, X, y=None):
        pct_null_threshold = self.parameters["pct_null_threshold"]
        X = infer_feature_types(X)
```

(continues on next page)

(continued from previous page)

```
        percent_null = X.isnull().mean()
        if pct_null_threshold == 0.0:
            null_cols = percent_null[percent_null > 0]
        else:
            null_cols = percent_null[percent_null >= pct_null_threshold]
        self._cols_to_drop = list(null_cols.index)
        return self

    def transform(self, X, y=None):
        """Transforms data X by dropping columns that exceed the threshold of null␣
↪values.
        Args:
            X (pd.DataFrame): Data to transform
            y (pd.Series, optional): Targets
        Returns:
            pd.DataFrame: Transformed X
        """

        X = infer_feature_types(X)
        return X.drop(columns=self._cols_to_drop)


pipeline_instance = BinaryClassificationPipeline(
    [
        "Imputer",
        MyDropNullColumns,
        "DateTime Featurizer",
        "Natural Language Featurizer",
        "One Hot Encoder",
        "Random Forest Classifier",
    ],
    custom_name="Pipeline with Custom Component",
    random_seed=20,
)

code = generate_pipeline_code(pipeline_instance)
print(code)

# This string can then be pasted into a separate window and run, although since the␣
↪pipeline has custom component `MyDropNullColumns`,
#      the code for that component must also be included
from evalml.demos import load_fraud

X, y = load_fraud(1000)
exec(code)
pipeline.fit(X, y)
```

```
from evalml.pipelines.binary_classification_pipeline import BinaryClassificationPipeline

pipeline = BinaryClassificationPipeline(
    component_graph={
        "Imputer": ["Imputer", "X", "y"],
```

(continues on next page)

(continued from previous page)

```
        "My Drop Null Columns Transformer": [MyDropNullColumns, "Imputer.x", "y"],
        "DateTime Featurizer": [
            "DateTime Featurizer",
            "My Drop Null Columns Transformer.x",
            "y",
        ],
        "Natural Language Featurizer": [
            "Natural Language Featurizer",
            "DateTime Featurizer.x",
            "y",
        ],
        "One Hot Encoder": ["One Hot Encoder", "Natural Language Featurizer.x", "y"],
        "Random Forest Classifier": [
            "Random Forest Classifier",
            "One Hot Encoder.x",
            "y",
        ],
    },
    parameters={
        "Imputer": {
            "categorical_impute_strategy": "most_frequent",
            "numeric_impute_strategy": "mean",
            "boolean_impute_strategy": "most_frequent",
            "categorical_fill_value": None,
            "numeric_fill_value": None,
            "boolean_fill_value": None,
        },
        "My Drop Null Columns Transformer": {"pct_null_threshold": 1.0},
        "DateTime Featurizer": {
            "features_to_extract": ["year", "month", "day_of_week", "hour"],
            "encode_as_categories": False,
            "time_index": None,
        },
        "One Hot Encoder": {
            "top_n": 10,
            "features_to_encode": None,
            "categories": None,
            "drop": "if_binary",
            "handle_unknown": "ignore",
            "handle_missing": "error",
        },
        "Random Forest Classifier": {"n_estimators": 100, "max_depth": 6, "n_jobs": -1},
    },
    custom_name="Pipeline with Custom Component",
    random_seed=20,
)

             Number of Features
Boolean                       1
Categorical                   6
Numeric                       5
```

(continues on next page)

(continued from previous page)

```
Number of training examples: 1000
Targets
False    85.90%
True     14.10%
Name: fraud, dtype: object
```

[25]:
```
pipeline = BinaryClassificationPipeline(component_graph={'Imputer': ['Imputer', 'X', 'y
↪'], 'My Drop Null Columns Transformer': [MyDropNullColumns, 'Imputer.x', 'y'],
↪'DateTime Featurizer': ['DateTime Featurizer', 'My Drop Null Columns Transformer.x', 'y
↪'], 'Natural Language Featurizer': ['Natural Language Featurizer', 'DateTime␣
↪Featurizer.x', 'y'], 'One Hot Encoder': ['One Hot Encoder', 'Natural Language␣
↪Featurizer.x', 'y'], 'Random Forest Classifier': ['Random Forest Classifier', 'One Hot␣
↪Encoder.x', 'y']}, parameters={'Imputer':{'categorical_impute_strategy': 'most_frequent
↪', 'numeric_impute_strategy': 'mean', 'boolean_impute_strategy': 'most_frequent',
↪'categorical_fill_value': None, 'numeric_fill_value': None, 'boolean_fill_value': None}
↪, 'My Drop Null Columns Transformer':{'pct_null_threshold': 1.0}, 'DateTime Featurizer
↪':{'features_to_extract': ['year', 'month', 'day_of_week', 'hour'], 'encode_as_
↪categories': False, 'time_index': None}, 'One Hot Encoder':{'top_n': 10, 'features_to_
↪encode': None, 'categories': None, 'drop': 'if_binary', 'handle_unknown': 'ignore',
↪'handle_missing': 'error'}, 'Random Forest Classifier':{'n_estimators': 100, 'max_depth
↪': 6, 'n_jobs': -1}}, custom_name='Pipeline with Custom Component', random_seed=20)
```

## 4.3 Component Graphs

EvalML component graphs represent and describe the flow of data in a collection of related components. A component graph is comprised of nodes representing components, and edges between pairs of nodes representing where the inputs and outputs of each component should go. It is the backbone of the features offered by the EvalML *pipeline*, but is also a powerful data structure on its own. EvalML currently supports component graphs as linear and directed acyclic graphs (DAG).

### 4.3.1 Defining a Component Graph

Component graphs can be defined by specifying the dictionary of components and edges that describe the graph.

In this dictionary, each key is a reference name for a component. Each corresponding value is a list, where the first element is the component itself, and the remaining elements are the input edges that should be connected to that component. The component as listed in the value can either be the component object itself or its string name.

This stucture is very similar to that of Dask computation graphs.

For example, in the code example below, we have a simple component graph made up of two components: an Imputer and a Random Forest Classifer. The names used to reference these two components are given by the keys, "My Imputer" and "RF Classifier" respectively. Each value in the dictionary is a list where the first element is the component corresponding to the component name, and the remaining elements are the inputs, e.g. "My Imputer" represents an Imputer component which has inputs "X" (the original features matrix) and "y" (the original target).

Feature edges are specified as `"X"` or `"{component_name}.x"`. For example, `{"My Component": [MyComponent, "Imputer.x", ...]}` indicates that we should use the feature output of the `Imputer` as as part of the feature input for MyComponent. Similarly, target edges are specified as `"y"` or `"{component_name}.y"`. `{"My Component": [MyComponent, "Target Imputer.y", ...]}` indicates that we should use the target output of the `Target Imputer` as a target input for MyComponent.

Each component can have a number of feature inputs, but can only have one target input. All input edges must be explicitly defined.

Using a real example, we define a simple component graph consisting of three nodes: an Imputer ("My Imputer"), an One-Hot Encoder ("OHE"), and a Random Forest Classifier ("RF Classifier").

- "My Imputer" takes the original X as a features input, and the original y as the target input
- "OHE" also takes the original X as a features input, and the original y as the target input
- "RF Classifer" takes the concatted feature outputs from "My Imputer" and "OHE" as a features input, and the original y as the target input.

[1]:
```
from evalml.pipelines import ComponentGraph

component_dict = {
    "My Imputer": ["Imputer", "X", "y"],
    "OHE": ["One Hot Encoder", "X", "y"],
    "RF Classifier": [
        "Random Forest Classifier",
        "My Imputer.x",
        "OHE.x",
        "y",
    ],  # takes in multiple feature inputs
}
cg_simple = ComponentGraph(component_dict)
```

All component graphs must end with one final or terminus node. This can either be a transformer or an estimator. Below, the component graph is invalid because has two terminus nodes: the "RF Classifier" and the "EN Classifier".

[2]:
```
# Can't instantiate a component graph with more than one terminus node (here: RF␣
↪Classifier, EN Classifier)
component_dict = {
    "My Imputer": ["Imputer", "X", "y"],
    "RF Classifier": ["Random Forest Classifier", "My Imputer.x", "y"],
    "EN Classifier": ["Elastic Net Classifier", "My Imputer.x", "y"],
}
```

Once we have defined a component graph, we can instantiate the graph with specific parameter values for each component using `.instantiate(parameters)`. All components in a component graph must be instantiated before fitting, transforming, or predicting.

Below, we instantiate our graph and set the value of our Imputer's `numeric_impute_strategy` to "most_frequent".

[3]:
```
cg_simple.instantiate({"My Imputer": {"numeric_impute_strategy": "most_frequent"}})
```

[3]:
```
{'My Imputer': ['Imputer', 'X', 'y'], 'OHE': ['One Hot Encoder', 'X', 'y'], 'RF␣
↪Classifier': ['Random Forest Classifier', 'My Imputer.x', 'OHE.x', 'y']}
```

### 4.3.2 Components in the Component Graph

You can use `.get_component(name)` and provide the unique component name to access any component in the component graph. Below, we can grab our Imputer component and confirm that `numeric_impute_strategy` has indeed been set to "most_frequent".

[4]:
```
cg_simple.get_component("My Imputer")
```

[4]:
```
Imputer(categorical_impute_strategy='most_frequent', numeric_impute_strategy='most_
↪frequent', boolean_impute_strategy='most_frequent', categorical_fill_value=None,␣
↪numeric_fill_value=None, boolean_fill_value=None)
```

You can also `.get_inputs(name)` and provide the unique component name to to retrieve all inputs for that component.

Below, we can grab our "RF Classifier" component and confirm that we use `"My Imputer.x"` as our features input and `"y"` as target input.

[5]:
```
cg_simple.get_inputs("RF Classifier")
```

[5]:
```
['My Imputer.x', 'OHE.x', 'y']
```

### 4.3.3 Component Graph Computation Order

Upon initalization, each component graph will generate a topological order. We can access this generated order by calling the `.compute_order` attribute. This attribute is used to determine the order that components should be evaluated during calls to `fit` and `transform`.

[6]:
```
cg_simple.compute_order
```

[6]:
```
['My Imputer', 'OHE', 'RF Classifier']
```

### 4.3.4 Visualizing Component Graphs

We can get more information about an instantiated component graph by calling `.describe()`. This method will pretty-print each of the components in the graph and its parameters.

[7]:
```
# Using a more involved component graph with more complex edges
component_dict = {
    "Imputer": ["Imputer", "X", "y"],
    "Target Imputer": ["Target Imputer", "X", "y"],
    "OneHot_RandomForest": ["One Hot Encoder", "Imputer.x", "Target Imputer.y"],
    "OneHot_ElasticNet": ["One Hot Encoder", "Imputer.x", "y"],
    "Random Forest": ["Random Forest Classifier", "OneHot_RandomForest.x", "y"],
    "Elastic Net": [
        "Elastic Net Classifier",
        "OneHot_ElasticNet.x",
        "Target Imputer.y",
    ],
    "Logistic Regression": [
        "Logistic Regression Classifier",
        "Random Forest.x",
        "Elastic Net.x",
        "y",
```

(continues on next page)

(continued from previous page)

```
    ],
}
cg_with_estimators = ComponentGraph(component_dict)
cg_with_estimators.instantiate({})
cg_with_estimators.describe()
```

```
1. Imputer
         * categorical_impute_strategy : most_frequent
         * numeric_impute_strategy : mean
         * boolean_impute_strategy : most_frequent
         * categorical_fill_value : None
         * numeric_fill_value : None
         * boolean_fill_value : None
2. Target Imputer
         * impute_strategy : most_frequent
         * fill_value : None
3. One Hot Encoder
         * top_n : 10
         * features_to_encode : None
         * categories : None
         * drop : if_binary
         * handle_unknown : ignore
         * handle_missing : error
4. One Hot Encoder
         * top_n : 10
         * features_to_encode : None
         * categories : None
         * drop : if_binary
         * handle_unknown : ignore
         * handle_missing : error
5. Random Forest Classifier
         * n_estimators : 100
         * max_depth : 6
         * n_jobs : -1
6. Elastic Net Classifier
         * penalty : elasticnet
         * C : 1.0
         * l1_ratio : 0.15
         * n_jobs : -1
         * multi_class : auto
         * solver : saga
7. Logistic Regression Classifier
         * penalty : l2
         * C : 1.0
         * n_jobs : -1
         * multi_class : auto
         * solver : lbfgs
```

We can also visualize a component graph by calling `.graph()`.

[8]:
```
cg_with_estimators.graph()
```

[8]:

### 4.3.5 Component graph methods

Similar to the pipeline structure, we can call `fit`, `transform` or `predict`.

We can also call `fit_features` which will fit all but the final component and `compute_final_component_features` which will transform all but the final component. These two methods may be useful in cases where you want to understand what transformed features are being passed into the last component.

[9]:
```
from evalml.demos import load_breast_cancer

X, y = load_breast_cancer()
component_dict = {
    "My Imputer": ["Imputer", "X", "y"],
    "OHE": ["One Hot Encoder", "My Imputer.x", "y"],
}
cg_with_final_transformer = ComponentGraph(component_dict)
cg_with_final_transformer.instantiate({})
cg_with_final_transformer.fit(X, y)

# We can call `transform` for ComponentGraphs with a final transformer
cg_with_final_transformer.transform(X, y)
```

```
         Number of Features
Numeric                  30

Number of training examples: 569
Targets
benign       62.74%
malignant    37.26%
Name: target, dtype: object
```

[9]:
```
     mean radius  mean texture  mean perimeter  mean area  mean smoothness  \
0          17.99         10.38          122.80     1001.0          0.11840
1          20.57         17.77          132.90     1326.0          0.08474
2          19.69         21.25          130.00     1203.0          0.10960
3          11.42         20.38           77.58      386.1          0.14250
4          20.29         14.34          135.10     1297.0          0.10030
..           ...           ...             ...        ...              ...
564        21.56         22.39          142.00     1479.0          0.11100
565        20.13         28.25          131.20     1261.0          0.09780
566        16.60         28.08          108.30      858.1          0.08455
567        20.60         29.33          140.10     1265.0          0.11780
568         7.76         24.54           47.92      181.0          0.05263

     mean compactness  mean concavity  mean concave points  mean symmetry  \
0             0.27760         0.30010              0.14710         0.2419
1             0.07864         0.08690              0.07017         0.1812
2             0.15990         0.19740              0.12790         0.2069
3             0.28390         0.24140              0.10520         0.2597
4             0.13280         0.19800              0.10430         0.1809
..                ...             ...                  ...            ...
```

(continues on next page)

(continued from previous page)

```
564           0.11590         0.24390              0.13890         0.1726
565           0.10340         0.14400              0.09791         0.1752
566           0.10230         0.09251              0.05302         0.1590
567           0.27700         0.35140              0.15200         0.2397
568           0.04362         0.00000              0.00000         0.1587

     mean fractal dimension  ...  worst radius  worst texture  \
0                   0.07871  ...        25.380          17.33
1                   0.05667  ...        24.990          23.41
2                   0.05999  ...        23.570          25.53
3                   0.09744  ...        14.910          26.50
4                   0.05883  ...        22.540          16.67
..                      ...  ...           ...            ...
564                 0.05623  ...        25.450          26.40
565                 0.05533  ...        23.690          38.25
566                 0.05648  ...        18.980          34.12
567                 0.07016  ...        25.740          39.42
568                 0.05884  ...         9.456          30.37

     worst perimeter  worst area  worst smoothness  worst compactness  \
0             184.60      2019.0           0.16220            0.66560
1             158.80      1956.0           0.12380            0.18660
2             152.50      1709.0           0.14440            0.42450
3              98.87       567.7           0.20980            0.86630
4             152.20      1575.0           0.13740            0.20500
..               ...         ...               ...                ...
564           166.10      2027.0           0.14100            0.21130
565           155.00      1731.0           0.11660            0.19220
566           126.70      1124.0           0.11390            0.30940
567           184.60      1821.0           0.16500            0.86810
568            59.16       268.6           0.08996            0.06444

     worst concavity  worst concave points  worst symmetry  \
0             0.7119                0.2654          0.4601
1             0.2416                0.1860          0.2750
2             0.4504                0.2430          0.3613
3             0.6869                0.2575          0.6638
4             0.4000                0.1625          0.2364
..               ...                   ...             ...
564           0.4107                0.2216          0.2060
565           0.3215                0.1628          0.2572
566           0.3403                0.1418          0.2218
567           0.9387                0.2650          0.4087
568           0.0000                0.0000          0.2871

     worst fractal dimension
0                    0.11890
1                    0.08902
2                    0.08758
3                    0.17300
4                    0.07678
..                       ...
```

(continues on next page)

(continued from previous page)

```
564                  0.07115
565                  0.06637
566                  0.07820
567                  0.12400
568                  0.07039

[569 rows x 30 columns]
```

[10]:
```
cg_with_estimators.fit(X, y)

# We can call `predict` for ComponentGraphs with a final transformer
cg_with_estimators.predict(X)
```

[10]:
```
0      malignant
1      malignant
2      malignant
3      malignant
4      malignant
         ...
564    malignant
565    malignant
566    malignant
567    malignant
568       benign
Length: 569, dtype: category
Categories (2, object): ['benign', 'malignant']
```

## 4.4 Components

Components are the lowest level of building blocks in EvalML. Each component represents a fundamental operation to be applied to data.

All components accept parameters as keyword arguments to their `__init__` methods. These parameters can be used to configure behavior.

Each component class definition must include a human-readable `name` for the component. Additionally, each component class may expose parameters for AutoML search by defining a `hyperparameter_ranges` attribute containing the parameters in question.

EvalML splits components into two categories: **transformers** and **estimators**.

### 4.4.1 Transformers

Transformers subclass the `Transformer` class, and define a `fit` method to learn information from training data and a `transform` method to apply a learned transformation to new data.

For example, an *imputer* is configured with the desired impute strategy to follow, for instance the mean value. The imputers `fit` method would learn the mean from the training data, and the `transform` method would fill the learned mean value in for any missing values in new data.

All transformers can execute `fit` and `transform` separately or in one step by calling `fit_transform`. Defining a custom `fit_transform` method can facilitate useful performance optimizations in some cases.

[1]:
```
import numpy as np
import pandas as pd
from evalml.pipelines.components import SimpleImputer

X = pd.DataFrame([[1, 2, 3], [1, np.nan, 3]])
display(X)
```

```
   0    1  2
0  1  2.0  3
1  1  NaN  3
```

[2]:
```
import woodwork as ww

imp = SimpleImputer(impute_strategy="mean")

X.ww.init()
X = imp.fit_transform(X)
display(X)
```

```
   0    1  2
0  1  2.0  3
1  1  2.0  3
```

Below is a list of all transformers included with EvalML:

[3]:
```
from evalml.pipelines.components.utils import all_components, Estimator, Transformer

for component in all_components():
    if issubclass(component, Transformer):
        print(f"Transformer: {component.name}")
```

```
Transformer: Time Series Regularizer
Transformer: Drop NaN Rows Transformer
Transformer: Replace Nullable Types Transformer
Transformer: Drop Rows Transformer
Transformer: URL Featurizer
Transformer: Email Featurizer
Transformer: Log Transformer
Transformer: STL Decomposer
Transformer: Polynomial Decomposer
Transformer: DFS Transformer
Transformer: Time Series Featurizer
Transformer: Natural Language Featurizer
Transformer: LSA Transformer
Transformer: Drop Null Columns Transformer
Transformer: DateTime Featurizer
Transformer: PCA Transformer
Transformer: Linear Discriminant Analysis Transformer
Transformer: Select Columns By Type Transformer
Transformer: Select Columns Transformer
Transformer: Drop Columns Transformer
Transformer: Oversampler
Transformer: Undersampler
Transformer: Standard Scaler
```

(continues on next page)

(continued from previous page)

```
Transformer: Time Series Imputer
Transformer: Target Imputer
Transformer: Imputer
Transformer: KNN Imputer
Transformer: Per Column Imputer
Transformer: Simple Imputer
Transformer: RFE Selector with RF Regressor
Transformer: RFE Selector with RF Classifier
Transformer: RF Regressor Select From Model
Transformer: RF Classifier Select From Model
Transformer: Ordinal Encoder
Transformer: Label Encoder
Transformer: Target Encoder
Transformer: One Hot Encoder
```

### 4.4.2 Estimators

Each estimator wraps an ML algorithm. Estimators subclass the `Estimator` class, and define a `fit` method to learn information from training data and a `predict` method for generating predictions from new data. Classification estimators should also define a `predict_proba` method for generating predicted probabilities.

Estimator classes each define a `model_family` attribute indicating what type of model is used.

Here's an example of using the *LogisticRegressionClassifier* estimator to fit and predict on a simple dataset:

[4]:
```
from evalml.pipelines.components import LogisticRegressionClassifier

clf = LogisticRegressionClassifier()

X = X
y = [1, 0]

clf.fit(X, y)
clf.predict(X)
```

[4]:
```
0    0
1    0
dtype: int64
```

Below is a list of all estimators included with EvalML:

[5]:
```
from evalml.pipelines.components.utils import all_components, Estimator, Transformer

for component in all_components():
    if issubclass(component, Estimator):
        print(f"Estimator: {component.name}")
```

```
Estimator: Stacked Ensemble Regressor
Estimator: Stacked Ensemble Classifier
Estimator: Vowpal Wabbit Regressor
Estimator: ARIMA Regressor
Estimator: Exponential Smoothing Regressor
Estimator: SVM Regressor
```

(continues on next page)

(continued from previous page)

```
Estimator: Prophet Regressor
Estimator: Time Series Baseline Estimator
Estimator: Decision Tree Regressor
Estimator: Baseline Regressor
Estimator: Extra Trees Regressor
Estimator: XGBoost Regressor
Estimator: CatBoost Regressor
Estimator: Random Forest Regressor
Estimator: LightGBM Regressor
Estimator: Linear Regressor
Estimator: Elastic Net Regressor
Estimator: Vowpal Wabbit Multiclass Classifier
Estimator: Vowpal Wabbit Binary Classifier
Estimator: SVM Classifier
Estimator: KNN Classifier
Estimator: Decision Tree Classifier
Estimator: LightGBM Classifier
Estimator: Baseline Classifier
Estimator: Extra Trees Classifier
Estimator: Elastic Net Classifier
Estimator: CatBoost Classifier
Estimator: XGBoost Classifier
Estimator: Random Forest Classifier
Estimator: Logistic Regression Classifier
```

### 4.4.3 Defining Custom Components

EvalML allows you to easily create your own custom components by following the steps below.

#### Custom Transformers

Your transformer must inherit from the correct subclass. In this case *Transformer* for components that transform data. Next we will use EvalML's *DropNullColumns* as an example.

[6]:
```
from evalml.pipelines.components import Transformer
from evalml.utils import (
    infer_feature_types,
)


class DropNullColumns(Transformer):
    """Transformer to drop features whose percentage of NaN values exceeds a specified␣
↪threshold"""

    name = "Drop Null Columns Transformer"
    hyperparameter_ranges = {}

    def __init__(self, pct_null_threshold=1.0, random_seed=0, **kwargs):
        """Initalizes an transformer to drop features whose percentage of NaN values␣
↪exceeds a specified threshold.
```

(continues on next page)

(continued from previous page)

```
        Args:
            pct_null_threshold(float): The percentage of NaN values in an input feature␣
↪to drop.
                Must be a value between [0, 1] inclusive. If equal to 0.0, will drop␣
↪columns with any null values.
                If equal to 1.0, will drop columns with all null values. Defaults to 0.
↪95.
        """
        if pct_null_threshold < 0 or pct_null_threshold > 1:
            raise ValueError(
                "pct_null_threshold must be a float between 0 and 1, inclusive."
            )
        parameters = {"pct_null_threshold": pct_null_threshold}
        parameters.update(kwargs)

        self._cols_to_drop = None
        super().__init__(
            parameters=parameters, component_obj=None, random_seed=random_seed
        )

    def fit(self, X, y=None):
        """Fits DropNullColumns component to data

        Args:
            X (pd.DataFrame): The input training data of shape [n_samples, n_features]
            y (pd.Series, optional): The target training data of length [n_samples]

        Returns:
            self
        """
        pct_null_threshold = self.parameters["pct_null_threshold"]
        X_t = infer_feature_types(X)
        percent_null = X_t.isnull().mean()
        if pct_null_threshold == 0.0:
            null_cols = percent_null[percent_null > 0]
        else:
            null_cols = percent_null[percent_null >= pct_null_threshold]
        self._cols_to_drop = list(null_cols.index)
        return self

    def transform(self, X, y=None):
        """Transforms data X by dropping columns that exceed the threshold of null␣
↪values.

        Args:
            X (pd.DataFrame): Data to transform
            y (pd.Series, optional): Ignored.

        Returns:
            pd.DataFrame: Transformed X
        """
```

(continues on next page)

(continued from previous page)

```
        X_t = infer_feature_types(X)
        return X_t.drop(self._cols_to_drop)
```

### Required fields

- `name`: A human-readable name.
- `modifies_features`: A boolean that specifies whether this component modifies (subsets or transforms) the features variable during `transform`.
- `modifies_target`: A boolean that specifies whether this component modifies (subsets or transforms) the target variable during `transform`.

### Required methods

Likewise, there are select methods you need to override as `Transformer` is an abstract base class:

- `__init__()`: The `__init__()` method of your transformer will need to call `super().__init__()` and pass three parameters in: a `parameters` dictionary holding the parameters to the component, the `component_obj`, and the `random_seed` value. You can see that `component_obj` is set to `None` above and we will discuss `component_obj` in depth later on.
- `fit()`: The `fit()` method is responsible for fitting your component on training data. It should return the component object.
- `transform()`: After fitting a component, the `transform()` method will take in new data and transform accordingly. It should return a pandas dataframe with woodwork initialized. Note: a component must call `fit()` before `transform()`.

You can also call or override `fit_transform()` that combines `fit()` and `transform()` into one method.

### Custom Estimators

Your estimator must inherit from the correct subclass. In this case *Estimator* for components that predict new target values. Next we will use EvalML's *BaselineRegressor* as an example.

[7]:
```
import numpy as np
import pandas as pd

from evalml.model_family import ModelFamily
from evalml.pipelines.components.estimators import Estimator
from evalml.problem_types import ProblemTypes


class BaselineRegressor(Estimator):
    """Regressor that predicts using the specified strategy.

    This is useful as a simple baseline regressor to compare with other regressors.
    """

    name = "Baseline Regressor"
    hyperparameter_ranges = {}
```

(continues on next page)

(continued from previous page)

```
    model_family = ModelFamily.BASELINE
    supported_problem_types = [
        ProblemTypes.REGRESSION,
        ProblemTypes.TIME_SERIES_REGRESSION,
    ]

    def __init__(self, strategy="mean", random_seed=0, **kwargs):
        """Baseline regressor that uses a simple strategy to make predictions.

        Args:
            strategy (str): Method used to predict. Valid options are "mean", "median".␣
↪Defaults to "mean".
            random_seed (int): Seed for the random number generator. Defaults to 0.

        """
        if strategy not in ["mean", "median"]:
            raise ValueError(
                "'strategy' parameter must equal either 'mean' or 'median'"
            )
        parameters = {"strategy": strategy}
        parameters.update(kwargs)

        self._prediction_value = None
        self._num_features = None
        super().__init__(
            parameters=parameters, component_obj=None, random_seed=random_seed
        )

    def fit(self, X, y=None):
        if y is None:
            raise ValueError("Cannot fit Baseline regressor if y is None")
        X = infer_feature_types(X)
        y = infer_feature_types(y)

        if self.parameters["strategy"] == "mean":
            self._prediction_value = y.mean()
        elif self.parameters["strategy"] == "median":
            self._prediction_value = y.median()
        self._num_features = X.shape[1]
        return self

    def predict(self, X):
        X = infer_feature_types(X)
        predictions = pd.Series([self._prediction_value] * len(X))
        return infer_feature_types(predictions)

    @property
    def feature_importance(self):
        """Returns importance associated with each feature. Since baseline regressors do␣
↪not use input features to calculate predictions, returns an array of zeroes.

        Returns:
```

(continues on next page)

(continued from previous page)

```
            np.ndarray (float): An array of zeroes

        """
        return np.zeros(self._num_features)
```

### Required fields

- `name`: A human-readable name.
- `model_family` - EvalML *model_family* that this component belongs to
- `supported_problem_types` - list of EvalML *problem_types* that this component supports
- `modifies_features`: A boolean that specifies whether the return value from `predict` or `predict_proba` should be used as features.
- `modifies_target`: A boolean that specifies whether the return value from `predict` or `predict_proba` should be used as the target variable.

Model families and problem types include:

[8]:
```
from evalml.model_family import ModelFamily
from evalml.problem_types import ProblemTypes

print("Model Families:\n", [m.value for m in ModelFamily])
print("Problem Types:\n", [p.value for p in ProblemTypes])
```

```
Model Families:
 ['k_neighbors', 'random_forest', 'svm', 'xgboost', 'lightgbm', 'linear_model', 'catboost
↪', 'extra_trees', 'ensemble', 'decision_tree', 'exponential_smoothing', 'arima',
↪'baseline', 'prophet', 'vowpal_wabbit', 'none']
Problem Types:
 ['binary', 'multiclass', 'regression', 'time series regression', 'time series binary',
↪'time series multiclass']
```

### Required methods

- `__init__()` - the `__init__()` method of your estimator will need to call `super().__init__()` and pass three parameters in: a `parameters` dictionary holding the parameters to the component, the `component_obj`, and the `random_seed` value.
- `fit()` - the `fit()` method is responsible for fitting your component on training data.
- `predict()` - after fitting a component, the `predict()` method will take in new data and predict new target values. Note: a component must call `fit()` before `predict()`.
- `feature_importance` - `feature_importance` is a Python property that returns a list of importances associated with each feature.

If your estimator handles classification problems it also requires an additonal method:

- `predict_proba()` - this method predicts probability estimates for classification labels

### Components Wrapping Third-Party Objects

The `component_obj` parameter is used for wrapping third-party objects and using them in component implementation. If you're using a `component_obj` you will need to define `__init__()` and pass in the relevant object that has also implemented the required methods mentioned above. However, if the `component_obj` does not follow EvalML component conventions, you may need to override methods as needed. Below is an example of EvalML's *LinearRegressor*.

[9]:
```
from sklearn.linear_model import LinearRegression as SKLinearRegression

from evalml.model_family import ModelFamily
from evalml.pipelines.components.estimators import Estimator
from evalml.problem_types import ProblemTypes


class LinearRegressor(Estimator):
    """Linear Regressor."""

    name = "Linear Regressor"
    model_family = ModelFamily.LINEAR_MODEL
    supported_problem_types = [ProblemTypes.REGRESSION]

    def __init__(
        self, fit_intercept=True, normalize=False, n_jobs=-1, random_seed=0, **kwargs
    ):
        parameters = {
            "fit_intercept": fit_intercept,
            "normalize": normalize,
            "n_jobs": n_jobs,
        }
        parameters.update(kwargs)
        linear_regressor = SKLinearRegression(**parameters)
        super().__init__(
            parameters=parameters,
            component_obj=linear_regressor,
            random_seed=random_seed,
        )

    @property
    def feature_importance(self):
        return self._component_obj.coef_
```

### Hyperparameter Ranges for AutoML

`hyperparameter_ranges` is a dictionary mapping the parameter name (str) to an allowed range (SkOpt Space) for that parameter. Both lists and `skopt.space.Categorical` values are accepted for categorical spaces.

AutoML will perform a search over the allowed ranges for each parameter to select models which produce optimal performance within those ranges. AutoML gets the allowed ranges for each component from the component's `hyperparameter_ranges` class attribute. Any component parameter you add an entry for in `hyperparameter_ranges` will be included in the AutoML search. If parameters are omitted, AutoML will use the default value in all pipelines.

### 4.4.4 Generate Component Code

Once you have a component defined in EvalML, you can generate string Python code to recreate this component, which can then be saved and run elsewhere with EvalML. `generate_component_code` requires a component instance as the input. This method works for custom components as well, although it won't return the code required to define the custom component.

[10]:
```
from evalml.pipelines.components import LogisticRegressionClassifier
from evalml.pipelines.components.utils import generate_component_code

lr = LogisticRegressionClassifier(C=5)
code = generate_component_code(lr)
print(code)
```

```
from evalml.pipelines.components.estimators.classifiers.logistic_regression_classifier␣
↪import LogisticRegressionClassifier

logisticRegressionClassifier = LogisticRegressionClassifier(**{'penalty': 'l2', 'C': 5,
↪'n_jobs': -1, 'multi_class': 'auto', 'solver': 'lbfgs'})
```

[11]:
```
# this string can then be copy and pasted into a separate window and executed as python␣
↪code
exec(code)
```

[12]:
```
# We can also do this for custom components
from evalml.pipelines.components.utils import generate_component_code

myDropNull = DropNullColumns()
print(generate_component_code(myDropNull))
```

```
dropNullColumnsTransformer = DropNullColumns(**{'pct_null_threshold': 1.0})
```

#### Expectations for Custom Classification Components

EvalML expects the following from custom classification component implementations:

- Classification targets will range from 0 to n-1 and are integers.
- For classification estimators, the order of predict_proba's columns must match the order of the target, and the column names must be integers ranging from 0 to n-1

## 4.5 Objectives

### 4.5.1 Overview

One of the key choices to make when training an ML model is what metric to choose by which to measure the efficacy of the model at learning the signal. Such metrics are useful for comparing how well the trained models generalize to new similar data.

This choice of metric is a key component of AutoML because it defines the cost function the AutoML search will seek to optimize. In EvalML, these metrics are called **objectives**. AutoML will seek to minimize (or maximize) the objective score as it explores more pipelines and parameters and will use the feedback from scoring pipelines to tune the

available hyperparameters and continue the search. Therefore, it is critical to have an objective function that represents how the model will be applied in the intended domain of use.

EvalML supports a variety of objectives from traditional supervised ML including mean squared error for regression problems and cross entropy or area under the ROC curve for classification problems. EvalML also allows the user to define a custom objective using their domain expertise, so that AutoML can search for models which provide the most value for the user's problem.

#### Optimization vs Ranking Objectives

There are many common objectives used for evaluating model performance. However, not all of these objectives should be used to optimize AutoMLSearch. Consider the popular objective `recall`, which is the number of true positives divided by the number of true positives and false negatives. If the model has no false negatives, the `recall` ends up being a perfect score of 1. During automatic optimization, models can exploit this by predicting the positive label in every case, making a completely useless but seemingly highly performant model. However, this objective is still useful when trying to evaluate performance after a model has been trained.

Due to this potential issue, we define two types of objectives: optimization and ranking. Optimization objectives are those that can be used within AutoMLSearch to train performant models. Ranking objectives can be used after AutoMLSearch has been run, to rank or otherwise evaluate model performance. These include all of the optimization metrics, as well as all other important metrics such as recall that are excluded from optimization.

Note that we also define a third class of objectives, non-core objectives, which are domain-specific and require additional configuration before they can be used.

### 4.5.2 Optimization Objectives

Use the `get_optimization_objectives` method to get a list of which objectives can be used for optimization in AutoMLSearch for each problem type:

[1]:
```
from evalml.objectives import get_optimization_objectives
from evalml.problem_types import ProblemTypes

for objective in get_optimization_objectives(ProblemTypes.BINARY):
    print(objective.name)
```

```
MCC Binary
Log Loss Binary
Gini
AUC
Precision
F1
Balanced Accuracy Binary
Accuracy Binary
```

### 4.5.3 Ranking Objectives

Use the `get_ranking_objectives` method to get a list of which objectives are included with EvalML for each problem type:

[2]:
```
from evalml.objectives import get_ranking_objectives

for objective in get_ranking_objectives(ProblemTypes.BINARY):
    print(objective.name)
```

```
MCC Binary
Log Loss Binary
Gini
AUC
Recall
Precision
F1
Balanced Accuracy Binary
Accuracy Binary
```

EvalML defines a base objective class for each problem type: `RegressionObjective`, `BinaryClassificationObjective` and `MulticlassClassificationObjective`. All EvalML objectives are a subclass of one of these.

#### Binary Classification Objectives and Thresholds

All binary classification objectives have a `threshold` property. Some binary classification objectives like log loss and AUC are unaffected by the choice of binary classification threshold, because they score based on predicted probabilities or examine a range of threshold values. These metrics are defined with `score_needs_proba` set to False. For all other binary classification objectives, we can compute the optimal binary classification threshold from the predicted probabilities and the target.

[3]:
```
from evalml.pipelines import BinaryClassificationPipeline
from evalml.demos import load_fraud
from evalml.objectives import F1

X, y = load_fraud(n_rows=100)
X.ww.init(
    logical_types={
        "provider": "Categorical",
        "region": "Categorical",
        "currency": "Categorical",
        "expiration_date": "Categorical",
    }
)
objective = F1()
pipeline = BinaryClassificationPipeline(
    component_graph=[
        "Imputer",
        "DateTime Featurizer",
        "One Hot Encoder",
        "Random Forest Classifier",
    ]
```

(continues on next page)

(continued from previous page)

```
)
pipeline.fit(X, y)
print(pipeline.threshold)
print(pipeline.score(X, y, objectives=[objective]))

y_pred_proba = pipeline.predict_proba(X)[True]
pipeline.threshold = objective.optimize_threshold(y_pred_proba, y)
print(pipeline.threshold)
print(pipeline.score(X, y, objectives=[objective]))
```

```
             Number of Features
Boolean                       1
Categorical                   6
Numeric                       5

Number of training examples: 100
Targets
False    91.00%
True      9.00%
Name: fraud, dtype: object
None
OrderedDict([('F1', 1.0)])
0.37905689607742854
OrderedDict([('F1', 1.0)])
```

### 4.5.4 Custom Objectives

Often times, the objective function is very specific to the use-case or business problem. To get the right objective to optimize requires thinking through the decisions or actions that will be taken using the model and assigning a cost/benefit to doing that correctly or incorrectly based on known outcomes in the training data.

Once you have determined the objective for your business, you can provide that to EvalML to optimize by defining a custom objective function.

#### Defining a Custom Objective Function

To create a custom objective class, we must define several elements:

- `name`: The printable name of this objective.
- `objective_function`: This function takes the predictions, true labels, and an optional reference to the inputs, and returns a score of how well the model performed.
- `greater_is_better`: `True` if a higher `objective_function` value represents a better solution, and otherwise `False`.
- `score_needs_proba`: Only for classification objectives. `True` if the objective is intended to function with predicted probabilities as opposed to predicted values (example: cross entropy for classifiers).
- `decision_function`: Only for binary classification objectives. This function takes predicted probabilities that were output from the model and a binary classification threshold, and returns predicted values.
- `perfect_score`: The score achieved by a perfect model on this objective.

- `expected_range`: The expected range of values we want this objective to output, which doesn't necessarily have to be equal to the possible range of values. For example, our expected R2 range is from `[-1, 1]`, although the actual range is `(-inf, 1]`.

### Example: Fraud Detection

To give a concrete example, let's look at how the *fraud detection* objective function is built.

[4]:
```
from evalml.objectives.binary_classification_objective import (
    BinaryClassificationObjective,
)
import pandas as pd


class FraudCost(BinaryClassificationObjective):
    """Score the percentage of money lost of the total transaction amount process due to␣
↪fraud"""

    name = "Fraud Cost"
    greater_is_better = False
    score_needs_proba = False
    perfect_score = 0.0

    def __init__(
        self,
        retry_percentage=0.5,
        interchange_fee=0.02,
        fraud_payout_percentage=1.0,
        amount_col="amount",
    ):
        """Create instance of FraudCost

        Args:
            retry_percentage (float): What percentage of customers that will retry a␣
↪transaction if it
                is declined. Between 0 and 1. Defaults to .5

            interchange_fee (float): How much of each successful transaction you can␣
↪collect.
                Between 0 and 1. Defaults to .02

            fraud_payout_percentage (float): Percentage of fraud you will not be able to␣
↪collect.
                Between 0 and 1. Defaults to 1.0

            amount_col (str): Name of column in data that contains the amount. Defaults␣
↪to "amount"
        """
        self.retry_percentage = retry_percentage
        self.interchange_fee = interchange_fee
        self.fraud_payout_percentage = fraud_payout_percentage
        self.amount_col = amount_col
```

(continues on next page)

(continued from previous page)

```
    def decision_function(self, ypred_proba, threshold=0.0, X=None):
        """Determine if a transaction is fraud given predicted probabilities, threshold,␣
↪and dataframe with transaction amount

        Args:
            ypred_proba (pd.Series): Predicted probablities
            X (pd.DataFrame): Dataframe containing transaction amount
            threshold (float): Dollar threshold to determine if transaction is fraud

        Returns:
            pd.Series: Series of predicted fraud labels using X and threshold
        """
        if not isinstance(X, pd.DataFrame):
            X = pd.DataFrame(X)

        if not isinstance(ypred_proba, pd.Series):
            ypred_proba = pd.Series(ypred_proba)

        transformed_probs = ypred_proba.values * X[self.amount_col]
        return transformed_probs > threshold

    def objective_function(self, y_true, y_predicted, X):
        """Calculate amount lost to fraud per transaction given predictions, true values,
↪ and dataframe with transaction amount

        Args:
            y_predicted (pd.Series): predicted fraud labels
            y_true (pd.Series): true fraud labels
            X (pd.DataFrame): dataframe with transaction amounts

        Returns:
            float: amount lost to fraud per transaction
        """
        if not isinstance(X, pd.DataFrame):
            X = pd.DataFrame(X)

        if not isinstance(y_predicted, pd.Series):
            y_predicted = pd.Series(y_predicted)

        if not isinstance(y_true, pd.Series):
            y_true = pd.Series(y_true)

        # extract transaction using the amount columns in users data
        try:
            transaction_amount = X[self.amount_col]
        except KeyError:
            raise ValueError("`{}` is not a valid column in X.".format(self.amount_col))

        # amount paid if transaction is fraud
        fraud_cost = transaction_amount * self.fraud_payout_percentage

        # money made from interchange fees on transaction
```

(continues on next page)

(continued from previous page)

```
        interchange_cost = (
            transaction_amount * (1 - self.retry_percentage) * self.interchange_fee
        )

        # calculate cost of missing fraudulent transactions
        false_negatives = (y_true & ~y_predicted) * fraud_cost

        # calculate money lost from fees
        false_positives = (~y_true & y_predicted) * interchange_cost

        loss = false_negatives.sum() + false_positives.sum()

        loss_per_total_processed = loss / transaction_amount.sum()

        return loss_per_total_processed
```

## 4.6 Model Understanding

Simply examining a model's performance metrics is not enough to select a model and promote it for use in a production setting. While developing an ML algorithm, it is important to understand how the model behaves on the data, to examine the key factors influencing its predictions and to consider where it may be deficient. Determination of what "success" may mean for an ML project depends first and foremost on the user's domain expertise.

EvalML includes a variety of tools for understanding models, from graphing utilities to methods for explaining predictions.

** Graphing methods on Jupyter Notebook and Jupyter Lab require ipywidgets to be installed.

** If graphing on Jupyter Lab, jupyterlab-plotly required. To download this, make sure you have npm installed.

### 4.6.1 Explaining Feature Influence

The EvalML package offers a variety of methods for understanding which features in a dataset have an impact on the output of the model. We can investigate this either through feature importance or through permutation importance, and leverage either in generating more readable explanations.

First, let's train a pipeline on some data.

[1]:
```
import evalml
from evalml.pipelines import BinaryClassificationPipeline

X, y = evalml.demos.load_breast_cancer()

X_train, X_holdout, y_train, y_holdout = evalml.preprocessing.split_data(
    X, y, problem_type="binary", test_size=0.2, random_seed=0
)


pipeline_binary = BinaryClassificationPipeline(
    component_graph={
        "Label Encoder": ["Label Encoder", "X", "y"],
```

(continues on next page)

(continued from previous page)

```
        "Imputer": ["Imputer", "X", "Label Encoder.y"],
        "Random Forest Classifier": [
            "Random Forest Classifier",
            "Imputer.x",
            "Label Encoder.y",
        ],
    }
)
pipeline_binary.fit(X_train, y_train)
print(pipeline_binary.score(X_holdout, y_holdout, objectives=["log loss binary"]))
```

```
         Number of Features
Numeric                  30

Number of training examples: 569
Targets
benign       62.74%
malignant    37.26%
Name: target, dtype: object
OrderedDict([('Log Loss Binary', 0.1686746297113362)])
```

### Feature Importance

We can get the importance associated with each feature of the resulting pipeline

[2]:
```
pipeline_binary.feature_importance
```

[2]:
```
                    feature  importance
0       mean concave points    0.138857
1           worst perimeter    0.137780
2      worst concave points    0.117782
3              worst radius    0.100584
4            mean concavity    0.086402
5                worst area    0.072027
6            mean perimeter    0.046500
7           worst concavity    0.043408
8               mean radius    0.037664
9                 mean area    0.033683
10             radius error    0.025036
11               area error    0.019324
12            worst texture    0.014754
13        worst compactness    0.014462
14             mean texture    0.013856
15         worst smoothness    0.013710
16           worst symmetry    0.011395
17          perimeter error    0.010284
18         mean compactness    0.008162
19          mean smoothness    0.008154
20  worst fractal dimension    0.007034
21  fractal dimension error    0.005502
22        compactness error    0.004953
23         smoothness error    0.004728
```

(continues on next page)

(continued from previous page)

```
24            texture error    0.004384
25           symmetry error    0.004250
26   mean fractal dimension    0.004164
27          concavity error    0.004089
28            mean symmetry    0.003997
29     concave points error    0.003076
```

We can also create a bar plot of the feature importances

[3]:
```
pipeline_binary.graph_feature_importance()
```

Data type cannot be displayed: application/vnd.plotly.v1+json, text/html

If we have a linear model, we can also view feature importance by simply inspecting the coefficients of the model.

[4]:
```
from evalml.model_understanding import get_linear_coefficients

pipeline_linear = BinaryClassificationPipeline(
    component_graph={
        "Label Encoder": ["Label Encoder", "X", "y"],
        "Imputer": ["Imputer", "X", "Label Encoder.y"],
        "Logistic Regression Classifier": [
            "Logistic Regression Classifier",
            "Imputer.x",
            "Label Encoder.y",
        ],
    }
)
pipeline_linear.fit(X_train, y_train)

get_linear_coefficients(pipeline_linear.estimator, features=X.columns)
```

[4]:
```
Intercept                 -0.352326
worst radius              -1.841560
mean radius               -1.734091
texture error             -0.769215
perimeter error           -0.301213
radius error              -0.078451
mean texture              -0.064297
mean perimeter            -0.041579
mean area                  0.001247
fractal dimension error    0.005983
smoothness error           0.006360
symmetry error             0.019811
mean fractal dimension     0.020884
worst area                 0.023366
concave points error       0.023432
compactness error          0.060427
mean smoothness            0.076231
concavity error            0.087974
mean symmetry              0.090586
```

(continues on next page)

(continued from previous page)

```
worst fractal dimension    0.102868
area error                 0.114724
worst smoothness           0.131197
mean concave points        0.190348
worst texture              0.251383
worst perimeter            0.284895
worst symmetry             0.285986
mean compactness           0.320826
worst concave points       0.361659
mean concavity             0.439937
worst compactness          0.981815
worst concavity            1.235671
dtype: float64
```

### Permutation Importance

We can also compute and plot the permutation importance of the pipeline.

[5]:
```
from evalml.model_understanding import calculate_permutation_importance

calculate_permutation_importance(
    pipeline_binary, X_holdout, y_holdout, "log loss binary"
)
```

[5]:
```
                    feature  importance
0           worst perimeter    0.063657
1                worst area    0.045759
2              worst radius    0.041926
3       mean concave points    0.029325
4      worst concave points    0.021045
5           worst concavity    0.010105
6             worst texture    0.010044
7              mean texture    0.006178
8             mean symmetry    0.005857
9                 mean area    0.004745
10         worst smoothness    0.003190
11               area error    0.003113
12           mean perimeter    0.002478
13   mean fractal dimension    0.001981
14        compactness error    0.001968
15          concavity error    0.001947
16            texture error    0.000291
17         smoothness error   -0.000206
18          mean smoothness   -0.000745
19  fractal dimension error   -0.000835
20        worst compactness   -0.002392
21           mean concavity   -0.003188
22         mean compactness   -0.005377
23             radius error   -0.006229
24              mean radius   -0.006870
25  worst fractal dimension   -0.007415
```

(continues on next page)

(continued from previous page)

```
26           symmetry error   -0.008175
27          perimeter error   -0.008980
28     concave points error   -0.010415
29           worst symmetry   -0.018645
```

[6]:
```
from evalml.model_understanding import graph_permutation_importance

graph_permutation_importance(pipeline_binary, X_holdout, y_holdout, "log loss binary")
```

Data type cannot be displayed: application/vnd.plotly.v1+json, text/html

### Human Readable Importance

We can generate a more human-comprehensible understanding of either the feature or permutation importance by using `readable_explanation(pipeline)`. This picks out a subset of features that have the highest impact on the output of the model, sorting them into either "heavily" or "somewhat" influential on the model. These features are selected either by feature importance or permutation importance with a given objective. If there are any features that actively decrease the performance of the pipeline, this function highlights those and recommends removal.

Note that permutation importance runs on the original input features, while feature importance runs on the features as they were passed in to the final estimator, having gone through a number of preprocessing steps. The two methods will highlight different features as being important, and feature names may vary as well.

[7]:
```
from evalml.model_understanding import readable_explanation

readable_explanation(
    pipeline_binary,
    X_holdout,
    y_holdout,
    objective="log loss binary",
    importance_method="permutation",
)
```

```
Random Forest Classifier: The output as measured by log loss binary is heavily␣
↪influenced by worst perimeter, and is somewhat influenced by worst area, worst radius,␣
↪mean concave points, and worst concave points.
The features smoothness error, mean smoothness, fractal dimension error, worst␣
↪compactness, mean concavity, mean compactness, radius error, mean radius, worst␣
↪fractal dimension, symmetry error, perimeter error, concave points error, and worst␣
↪symmetry detracted from model performance. We suggest removing these features.
```

[8]:
```
readable_explanation(
    pipeline_binary, importance_method="feature"
)  # feature importance doesn't require X and y
```

```
Random Forest Classifier: The output is somewhat influenced by mean concave points,␣
↪worst perimeter, worst concave points, worst radius, and mean concavity.
```

We can adjust the number of most important features visible with the `max_features` argument, or modify the minimum threshold for "importance" with `min_importance_threshold`. However, these values will not affect any detrimental features displayed, as this function always displays all of them.

### 4.6.2 Metrics for Model Understanding

#### Confusion Matrix

For binary or multiclass classification, we can view a confusion matrix of the classifier's predictions. In the DataFrame output of `confusion_matrix()`, the column header represents the predicted labels while row header represents the actual labels.

[9]:
```
from evalml.model_understanding.metrics import confusion_matrix

y_pred = pipeline_binary.predict(X_holdout)
confusion_matrix(y_holdout, y_pred)
```

[9]:
```
             benign  malignant
benign     0.930556   0.069444
malignant  0.023810   0.976190
```

[10]:
```
from evalml.model_understanding.metrics import graph_confusion_matrix

y_pred = pipeline_binary.predict(X_holdout)
graph_confusion_matrix(y_holdout, y_pred)
```

Data type cannot be displayed: application/vnd.plotly.v1+json, text/html

#### Precision-Recall Curve

For binary classification, we can view the precision-recall curve of the pipeline.

[11]:
```
from evalml.model_understanding.metrics import graph_precision_recall_curve

# get the predicted probabilities associated with the "true" label
import woodwork as ww

y_encoded = y_holdout.ww.map({"benign": 0, "malignant": 1})
y_pred_proba = pipeline_binary.predict_proba(X_holdout)["malignant"]
graph_precision_recall_curve(y_encoded, y_pred_proba)
```

Data type cannot be displayed: application/vnd.plotly.v1+json, text/html

#### ROC Curve

For binary and multiclass classification, we can view the Receiver Operating Characteristic (ROC) curve of the pipeline.

[12]:
```
from evalml.model_understanding.metrics import graph_roc_curve

# get the predicted probabilities associated with the "malignant" label
y_pred_proba = pipeline_binary.predict_proba(X_holdout)["malignant"]
graph_roc_curve(y_encoded, y_pred_proba)
```

Data type cannot be displayed: application/vnd.plotly.v1+json, text/html

The ROC curve can also be generated for multiclass classification problems. For multiclass problems, the graph will show a one-vs-many ROC curve for each class.

[13]:
```
from evalml.pipelines import MulticlassClassificationPipeline

X_multi, y_multi = evalml.demos.load_wine()

pipeline_multi = MulticlassClassificationPipeline(
    ["Simple Imputer", "Random Forest Classifier"]
)
pipeline_multi.fit(X_multi, y_multi)

y_pred_proba = pipeline_multi.predict_proba(X_multi)
graph_roc_curve(y_multi, y_pred_proba)
```

```
         Number of Features
Numeric                  13

Number of training examples: 178
Targets
class_1    39.89%
class_0    33.15%
class_2    26.97%
Name: target, dtype: object
```

Data type cannot be displayed: application/vnd.plotly.v1+json, text/html

### 4.6.3 Visualizations

#### Binary Objective Score vs. Threshold Graph

Some *binary classification objectives* (objectives that have `score_needs_proba` set to False) are sensitive to a decision threshold. For those objectives, we can obtain and graph the scores for thresholds from zero to one, calculated at evenly-spaced intervals determined by `steps`.

[14]:
```
from evalml.model_understanding.visualizations import binary_objective_vs_threshold

binary_objective_vs_threshold(pipeline_binary, X_holdout, y_holdout, "f1", steps=10)
```

[14]:
```
    threshold     score
0         0.0  0.538462
1         0.1  0.811881
2         0.2  0.891304
3         0.3  0.901099
4         0.4  0.931818
5         0.5  0.931818
6         0.6  0.941176
```

(continues on next page)

(continued from previous page)

```
7          0.7  0.951220
8          0.8  0.936709
9          0.9  0.923077
10         1.0  0.000000
```

[15]:
```
from evalml.model_understanding.visualizations import (
    graph_binary_objective_vs_threshold,
)

graph_binary_objective_vs_threshold(
    pipeline_binary, X_holdout, y_holdout, "f1", steps=100
)
```

Data type cannot be displayed: application/vnd.plotly.v1+json, text/html

### Predicted Vs Actual Values Graph for Regression Problems

We can also create a scatterplot comparing predicted vs actual values for regression problems. We can specify an `outlier_threshold` to color values differently if the absolute difference between the actual and predicted values are outside of a given threshold.

[16]:
```
from evalml.model_understanding.visualizations import graph_prediction_vs_actual
from evalml.pipelines import RegressionPipeline

X_regress, y_regress = evalml.demos.load_diabetes()
X_train_reg, X_test_reg, y_train_reg, y_test_reg = evalml.preprocessing.split_data(
    X_regress, y_regress, problem_type="regression"
)

pipeline_regress = RegressionPipeline(["One Hot Encoder", "Linear Regressor"])
pipeline_regress.fit(X_train_reg, y_train_reg)

y_pred = pipeline_regress.predict(X_test_reg)
graph_prediction_vs_actual(y_test_reg, y_pred, outlier_threshold=50)
```

```
         Number of Features
Numeric                  10

Number of training examples: 442
Targets
200    1.36%
72     1.36%
90     1.13%
178    1.13%
71     1.13%
       ...
73     0.23%
222    0.23%
86     0.23%
79     0.23%
```

(continues on next page)

(continued from previous page)

```
57     0.23%
Name: target, Length: 214, dtype: object
```

Data type cannot be displayed: application/vnd.plotly.v1+json, text/html

### Tree Visualization

Now let's train a decision tree on some data. We can visualize the structure of the Decision Tree that was fit to that data, and save it if necessary.

[17]:
```
pipeline_dt = BinaryClassificationPipeline(
    ["Simple Imputer", "Decision Tree Classifier"]
)
pipeline_dt.fit(X_train, y_train)
```

[17]:
```
pipeline = BinaryClassificationPipeline(component_graph={'Simple Imputer': ['Simple␣
↪Imputer', 'X', 'y'], 'Decision Tree Classifier': ['Decision Tree Classifier', 'Simple␣
↪Imputer.x', 'y']}, parameters={'Simple Imputer':{'impute_strategy': 'most_frequent',
↪'fill_value': None}, 'Decision Tree Classifier':{'criterion': 'gini', 'max_features':
↪'auto', 'max_depth': 6, 'min_samples_split': 2, 'min_weight_fraction_leaf': 0.0}},␣
↪random_seed=0)
```

[18]:
```
from evalml.model_understanding.visualizations import visualize_decision_tree

visualize_decision_tree(
    pipeline_dt.estimator, max_depth=2, rotate=False, filled=True, filepath=None
)
```

[18]:

### Confusion Matrix and Thresholds for Binary Classification Pipelines

For binary classification pipelines, EvalML also provides the ability to compare the actual positive and actual negative histograms, as well as obtaining the confusion matrices and ideal thresholds per objective.

[19]:
```
from evalml.model_understanding import find_confusion_matrix_per_thresholds

df, objective_thresholds = find_confusion_matrix_per_thresholds(
    pipeline_binary, X, y, n_bins=10
)
df.head(10)
```

[19]:
```
     true_pos_count  true_neg_count  true_positives  true_negatives  \
0.1               1             309             211             309
0.2               0              35             211             344
0.3               0               5             211             349
0.4               0               3             211             352
0.5               0               0             211             352
0.6               3               2             208             354
0.7               2               2             206             356
0.8               9               1             197             357
```

(continues on next page)

(continued from previous page)

```
0.9              15               0             182             357
1.0             182               0               0             357

     false_positives  false_negatives               data_in_bins
0.1               48                1       [19, 20, 21, 37, 46]
0.2               13                1    [68, 92, 123, 133, 147]
0.3                8                1  [112, 157, 484, 491, 505]
0.4                5                1            [208, 340, 465]
0.5                5                1                         []
0.6                3                4    [40, 89, 128, 263, 297]
0.7                1                6         [13, 81, 385, 421]
0.8                0               15       [38, 41, 54, 73, 86]
0.9                0               30      [39, 44, 91, 99, 100]
1.0                0              212            [0, 1, 2, 3, 4]
```

[20]:
```
objective_thresholds
```

[20]:
```
{'accuracy': {'objective score': 0.9894551845342706, 'threshold value': 0.4},
 'balanced_accuracy': {'objective score': 0.9906387083135141,
  'threshold value': 0.4},
 'precision': {'objective score': 1.0, 'threshold value': 0.8},
 'f1': {'objective score': 0.9859813084112149, 'threshold value': 0.4}}
```

In the above results, the first dataframe contains the histograms for the actual positive and negative classes, indicated by `true_pos_count` and `true_neg_count`. The columns `true_positives`, `true_negatives`, `false_positives`, and `false_negatives` contain the confusion matrix information for the associated threshold, and the `data_in_bins` holds a random subset of row indices (both postive and negative) that belong in each bin. The index of the dataframe represents the associated threshold. For instance, at index `0.1`, there is 1 positive and 309 negative rows that fall between `[0.0, 0.1]`.

The returned `objective_thresholds` dictionary has the objective measure as the key, and the dictionary value associated contains both the best objective score and the threshold that results in the associated score.

### Visualize high dimensional data in lower space

We can use T-SNE to visualize data with many features on a 2D plot, making it easier to see relationships in your data.

[21]:
```
# Our data is highly dimensional, we can't plot this in a way we understand
print(len(X.columns))
```

```
30
```

[22]:
```
from evalml.model_understanding import graph_t_sne

fig = graph_t_sne(X)
fig
```

Data type cannot be displayed: application/vnd.plotly.v1+json, text/html

### 4.6.4 Partial Dependence Plots

We can calculate the one-way partial dependence plots for a feature.

[23]:
```
from evalml.model_understanding import partial_dependence

partial_dependence(
    pipeline_binary, X_holdout, features="mean radius", grid_resolution=5
)
```

[23]:
```
   feature_values  partial_dependence class_label
0         9.69092            0.392453   malignant
1        12.40459            0.395962   malignant
2        15.11826            0.417396   malignant
3        17.83193            0.429542   malignant
4        20.54560            0.429717   malignant
```

[24]:
```
from evalml.model_understanding import graph_partial_dependence

graph_partial_dependence(
    pipeline_binary, X_holdout, features="mean radius", grid_resolution=5
)
```

Data type cannot be displayed: application/vnd.plotly.v1+json, text/html

We can also compute the partial dependence for a categorical feature. We will demonstrate this on the fraud dataset.

[25]:
```
X_fraud, y_fraud = evalml.demos.load_fraud(100, verbose=False)
X_fraud.ww.init(
    logical_types={
        "provider": "Categorical",
        "region": "Categorical",
        "currency": "Categorical",
        "expiration_date": "Categorical",
    }
)

fraud_pipeline = BinaryClassificationPipeline(
    ["DateTime Featurizer", "One Hot Encoder", "Random Forest Classifier"]
)
fraud_pipeline.fit(X_fraud, y_fraud)

graph_partial_dependence(fraud_pipeline, X_fraud, features="provider")
```

Data type cannot be displayed: application/vnd.plotly.v1+json, text/html

Two-way partial dependence plots are also possible and invoke the same API.

[26]:
```
partial_dependence(
    pipeline_binary,
    X_holdout,
```

(continues on next page)

(continued from previous page)

```
    features=("worst perimeter", "worst radius"),
    grid_resolution=5,
)
```

[26]:
```
             10.6876  14.404924999999999  18.12225  21.839575   25.5569  \
69.140700   0.279038            0.282898  0.435179   0.435355  0.435355
94.334275   0.304335            0.308194  0.458283   0.458458  0.458458
119.527850  0.464455            0.468314  0.612137   0.616932  0.616932
144.721425  0.483437            0.487297  0.631120   0.635915  0.635915
169.915000  0.483437            0.487297  0.631120   0.635915  0.635915

           class_label
69.140700    malignant
94.334275    malignant
119.527850   malignant
144.721425   malignant
169.915000   malignant
```

[27]:
```
graph_partial_dependence(
    pipeline_binary,
    X_holdout,
    features=("worst perimeter", "worst radius"),
    grid_resolution=5,
)
```

Data type cannot be displayed: application/vnd.plotly.v1+json, text/html

### 4.6.5 Explaining Predictions

We can explain why the model made certain predictions with the *explain_predictions* function. This can use either the Shapley Additive Explanations (SHAP) algorithm or the Local Interpretable Model-agnostic Explanations (LIME) algorithm to identify the top features that explain the predicted value.

This function can explain both classification and regression models - all you need to do is provide the pipeline, the input features, and a list of rows corresponding to the indices of the input features you want to explain. The function will return a table that you can print summarizing the top 3 most positive and negative contributing features to the predicted value.

In the example below, we explain the prediction for the third data point in the data set. We see that the `worst concave points` feature increased the estimated probability that the tumor is malignant by 20% while the `worst radius` feature decreased the probability the tumor is malignant by 5%.

[28]:
```
from evalml.model_understanding.prediction_explanations import explain_predictions

table = explain_predictions(
    pipeline=pipeline_binary,
    input_features=X_holdout,
    y=None,
    indices_to_explain=[3],
    top_k_features=6,
```

(continues on next page)

(continued from previous page)

```
    include_explainer_values=True,
)
print(table)
```

```
Random Forest Classifier w/ Label Encoder + Imputer

{'Label Encoder': {'positive_label': None}, 'Imputer': {'categorical_impute_strategy':
↪'most_frequent', 'numeric_impute_strategy': 'mean', 'boolean_impute_strategy': 'most_
↪frequent', 'categorical_fill_value': None, 'numeric_fill_value': None, 'boolean_fill_
↪value': None}, 'Random Forest Classifier': {'n_estimators': 100, 'max_depth': 6, 'n_
↪jobs': -1}}

        1 of 1

                   Feature Name       Feature Value   Contribution to Prediction   SHAP␣
↪Value
               ␣
↪==============================================================================
                 worst concavity         0.18                    -                  -0.
↪02
                 mean concavity          0.04                    -                  -0.
↪03
                   worst area           599.50                   -                  -0.
↪03
                  worst radius           14.04                   -                  -0.
↪05
               mean concave points       0.03                    -                  -0.
↪05
                 worst perimeter         92.80                   -                  -0.
↪06
```

The interpretation of the table is the same for regression problems - but the SHAP value now corresponds to the change in the estimated value of the dependent variable rather than a change in probability. For multiclass classification problems, a table will be output for each possible class.

Below is an example of how you would explain three predictions with *explain_predictions*.

[29]:
```
from evalml.model_understanding.prediction_explanations import explain_predictions

report = explain_predictions(
    pipeline=pipeline_binary,
    input_features=X_holdout,
    y=y_holdout,
    indices_to_explain=[0, 4, 9],
    include_explainer_values=True,
    output_format="text",
)
print(report)
```

```
Random Forest Classifier w/ Label Encoder + Imputer
```

(continues on next page)

(continued from previous page)

```
{'Label Encoder': {'positive_label': None}, 'Imputer': {'categorical_impute_strategy':
↪'most_frequent', 'numeric_impute_strategy': 'mean', 'boolean_impute_strategy': 'most_
↪frequent', 'categorical_fill_value': None, 'numeric_fill_value': None, 'boolean_fill_
↪value': None}, 'Random Forest Classifier': {'n_estimators': 100, 'max_depth': 6, 'n_
↪jobs': -1}}

        1 of 3

                    Feature Name       Feature Value   Contribution to Prediction   SHAP␣
↪Value
               ␣
↪==============================================================================
                  worst perimeter         101.20                    -                  -0.
↪04
                worst concave points       0.06                     -                  -0.
↪05
                mean concave points        0.01                     -                  -0.
↪05


        2 of 3

                    Feature Name       Feature Value   Contribution to Prediction   SHAP␣
↪Value
               ␣
↪===============================================================================
                   worst radius           11.94                    -                  -0.
↪05
                  worst perimeter         80.78                    -                  -0.
↪06
                mean concave points       0.02                     -                  -0.
↪06


        3 of 3

                    Feature Name       Feature Value   Contribution to Prediction   SHAP␣
↪Value
               ␣
↪==============================================================================
                worst concave points       0.10                     -                  -0.
↪05
                  worst perimeter          99.21                    -                  -0.
↪06
                mean concave points        0.03                     -                  -0.
↪08
```

The above examples used the SHAP algorithm, since that is what `explain_predictions` uses by default. If you would like to use LIME instead, you can change that with the `algorithm="lime"` argument.

[30]:
```
from evalml.model_understanding.prediction_explanations import explain_predictions

table = explain_predictions(
    pipeline=pipeline_binary,
    input_features=X_holdout,
    y=None,
    indices_to_explain=[3],
    top_k_features=6,
    include_explainer_values=True,
    algorithm="lime",
)
print(table)
```

```
Random Forest Classifier w/ Label Encoder + Imputer

{'Label Encoder': {'positive_label': None}, 'Imputer': {'categorical_impute_strategy':
↪'most_frequent', 'numeric_impute_strategy': 'mean', 'boolean_impute_strategy': 'most_
↪frequent', 'categorical_fill_value': None, 'numeric_fill_value': None, 'boolean_fill_
↪value': None}, 'Random Forest Classifier': {'n_estimators': 100, 'max_depth': 6, 'n_
↪jobs': -1}}

        1 of 1

                    Feature Name       Feature Value   Contribution to Prediction   LIME␣
↪Value
              ␣
↪==============================================================================
                    worst radius           14.04                    +                   0.
↪06
                  worst perimeter          92.80                    +                   0.
↪06
                     worst area           599.50                    +                   0.
↪05
                mean concave points        0.03                     +                   0.
↪05
                worst concave points       0.12                     +                   0.
↪04
                  worst concavity          0.18                     +                   0.
↪03
```

[31]:
```
from evalml.model_understanding.prediction_explanations import explain_predictions

report = explain_predictions(
    pipeline=pipeline_binary,
    input_features=X_holdout,
    y=None,
    indices_to_explain=[0, 4, 9],
    include_explainer_values=True,
    output_format="text",
    algorithm="lime",
```

(continues on next page)

(continued from previous page)

```
)
print(report)
```

```
Random Forest Classifier w/ Label Encoder + Imputer

{'Label Encoder': {'positive_label': None}, 'Imputer': {'categorical_impute_strategy':
↪'most_frequent', 'numeric_impute_strategy': 'mean', 'boolean_impute_strategy': 'most_
↪frequent', 'categorical_fill_value': None, 'numeric_fill_value': None, 'boolean_fill_
↪value': None}, 'Random Forest Classifier': {'n_estimators': 100, 'max_depth': 6, 'n_
↪jobs': -1}}

        1 of 3

                 Feature Name     Feature Value   Contribution to Prediction   LIME Value
                ==========================================================================
                worst perimeter      101.20                   +                    0.06
                 worst radius         15.14                   +                    0.06
                  worst area         718.90                   +                    0.05


        2 of 3

                 Feature Name     Feature Value   Contribution to Prediction   LIME Value
                ==========================================================================
                worst perimeter       80.78                   +                    0.06
                 worst radius         11.94                   +                    0.06
                  worst area         433.10                   +                    0.05


        3 of 3

                 Feature Name     Feature Value   Contribution to Prediction   LIME Value
                ==========================================================================
                 worst radius         14.42                   +                    0.06
                worst perimeter       99.21                   +                    0.06
                  worst area         634.30                   +                    0.05
```

### Explaining Best and Worst Predictions

When debugging machine learning models, it is often useful to analyze the best and worst predictions the model made. The *explain_predictions_best_worst* function can help us with this.

This function will display the output of *explain_predictions* for the best 2 and worst 2 predictions. By default, the best and worst predictions are determined by the absolute error for regression problems and cross entropy for classification problems.

We can specify our own ranking function by passing in a function to the `metric` parameter. This function will be called on `y_true` and `y_pred`. By convention, lower scores are better.

At the top of each table, we can see the predicted probabilities, target value, error, and row index for that prediction. For a regression problem, we would see the predicted value instead of predicted probabilities.

[32]:
```
from evalml.model_understanding.prediction_explanations import (
    explain_predictions_best_worst,
)

shap_report = explain_predictions_best_worst(
    pipeline=pipeline_binary,
    input_features=X_holdout,
    y_true=y_holdout,
    include_explainer_values=True,
    top_k_features=6,
    num_to_explain=2,
)

print(shap_report)
```

```
Random Forest Classifier w/ Label Encoder + Imputer

{'Label Encoder': {'positive_label': None}, 'Imputer': {'categorical_impute_strategy':
↪'most_frequent', 'numeric_impute_strategy': 'mean', 'boolean_impute_strategy': 'most_
↪frequent', 'categorical_fill_value': None, 'numeric_fill_value': None, 'boolean_fill_
↪value': None}, 'Random Forest Classifier': {'n_estimators': 100, 'max_depth': 6, 'n_
↪jobs': -1}}

        Best 1 of 2

                Predicted Probabilities: [benign: 1.0, malignant: 0.0]
                Predicted Value: benign
                Target Value: benign
                Cross Entropy: 0.0
                Index ID: 502

                    Feature Name       Feature Value   Contribution to Prediction   SHAP␣
↪Value
                ␣
↪==========================================================================
                   mean concavity          0.06                    -                  -0.
↪03
                     worst area           552.00                   -                  -0.
↪03
                worst concave points       0.08                    -                  -0.
↪05
                    worst radius           13.57                   -                  -0.
↪05
                mean concave points        0.03                    -                  -0.
↪05
                  worst perimeter          86.67                   -                  -0.
↪06


        Best 2 of 2

                Predicted Probabilities: [benign: 1.0, malignant: 0.0]
                Predicted Value: benign
```

(continues on next page)

(continued from previous page)

```
                Target Value: benign
                Cross Entropy: 0.0
                Index ID: 52

                    Feature Name       Feature Value   Contribution to Prediction   SHAP␣
↪Value
                ␣
↪==============================================================================
                   mean concavity          0.02                    -                  -0.
↪02
                     worst area           527.20                   -                  -0.
↪03
                    worst radius           13.10                   -                  -0.
↪04
                worst concave points       0.06                    -                  -0.
↪04
                mean concave points        0.01                    -                  -0.
↪05
                  worst perimeter          83.67                   -                  -0.
↪06


        Worst 1 of 2

                Predicted Probabilities: [benign: 0.266, malignant: 0.734]
                Predicted Value: malignant
                Target Value: benign
                Cross Entropy: 1.325
                Index ID: 363

                 Feature Name     Feature Value   Contribution to Prediction   SHAP Value
                ==============================================================================
                worst perimeter      117.20                  +                   0.13
                 worst radius         18.13                  +                   0.12
                  worst area         1009.00                 +                   0.11
                   mean area         838.10                  +                   0.06
                  mean radius         16.50                  +                   0.05
                worst concavity       0.17                   -                  -0.05


        Worst 2 of 2

                Predicted Probabilities: [benign: 1.0, malignant: 0.0]
                Predicted Value: benign
                Target Value: malignant
                Cross Entropy: 7.987
                Index ID: 135

                    Feature Name       Feature Value   Contribution to Prediction   SHAP␣
↪Value
                ␣
↪==============================================================================
```

(continues on next page)

(continued from previous page)

```
                    mean concavity          0.05                    -                    -0.
↪03
                      worst area           653.60                   -                    -0.
↪04
                 worst concave points       0.09                    -                    -0.
↪05
                     worst radius           14.49                   -                    -0.
↪05
                   worst perimeter          92.04                   -                    -0.
↪06
                 mean concave points        0.03                    -                    -0.
↪06
```

[33]:
```
lime_report = explain_predictions_best_worst(
    pipeline=pipeline_binary,
    input_features=X_holdout,
    y_true=y_holdout,
    include_explainer_values=True,
    top_k_features=6,
    num_to_explain=2,
    algorithm="lime",
)

print(lime_report)
```

```
Random Forest Classifier w/ Label Encoder + Imputer

{'Label Encoder': {'positive_label': None}, 'Imputer': {'categorical_impute_strategy':
↪'most_frequent', 'numeric_impute_strategy': 'mean', 'boolean_impute_strategy': 'most_
↪frequent', 'categorical_fill_value': None, 'numeric_fill_value': None, 'boolean_fill_
↪value': None}, 'Random Forest Classifier': {'n_estimators': 100, 'max_depth': 6, 'n_
↪jobs': -1}}

        Best 1 of 2

                Predicted Probabilities: [benign: 1.0, malignant: 0.0]
                Predicted Value: benign
                Target Value: benign
                Cross Entropy: 0.0
                Index ID: 502

                    Feature Name       Feature Value   Contribution to Prediction   LIME␣
↪Value
                ␣
↪==============================================================================
                   worst perimeter          86.67                   +                    0.
↪06
                     worst radius           13.57                   +                    0.
↪06
```

(continues on next page)

(continued from previous page)

```
                    worst area           552.00                    +                    0.
↪05
                mean concave points        0.03                    +                    0.
↪04
                worst concave points       0.08                    +                    0.
↪04
                  worst concavity          0.19                    +                    0.
↪03


        Best 2 of 2

                Predicted Probabilities: [benign: 1.0, malignant: 0.0]
                Predicted Value: benign
                Target Value: benign
                Cross Entropy: 0.0
                Index ID: 52

                    Feature Name       Feature Value   Contribution to Prediction   LIME␣
↪Value
               ␣
↪==============================================================================
                    worst radius           13.10                    +                    0.
↪06
                  worst perimeter          83.67                    +                    0.
↪06
                    worst area           527.20                    +                    0.
↪05
                mean concave points        0.01                    +                    0.
↪04
                worst concave points       0.06                    +                    0.
↪04
                  worst concavity          0.09                    +                    0.
↪03


        Worst 1 of 2

                Predicted Probabilities: [benign: 0.266, malignant: 0.734]
                Predicted Value: malignant
                Target Value: benign
                Cross Entropy: 1.325
                Index ID: 363

                    Feature Name       Feature Value   Contribution to Prediction   LIME␣
↪Value
               ␣
↪==============================================================================
                  worst concavity          0.17                    -                   -0.
↪03
                worst concave points       0.09                    -                   -0.
↪04
```

(continues on next page)

(continued from previous page)

```
                mean concave points        0.05                    -                   -0.
↪04
                     worst area           1009.00                  -                   -0.
↪05
                   worst radius            18.13                   -                   -0.
↪06
                 worst perimeter          117.20                   -                   -0.
↪06


        Worst 2 of 2

                Predicted Probabilities: [benign: 1.0, malignant: 0.0]
                Predicted Value: benign
                Target Value: malignant
                Cross Entropy: 7.987
                Index ID: 135

                    Feature Name       Feature Value   Contribution to Prediction   LIME␣
↪Value
               ␣
↪==========================================================================
                 worst perimeter          92.04                   +                   0.
↪06
                   worst radius           14.49                   +                   0.
↪06
                     worst area          653.60                   +                   0.
↪05
                mean concave points        0.03                   +                   0.
↪04
                worst concave points       0.09                   +                   0.
↪04
                  worst concavity          0.22                   +                   0.
↪03
```

We use a custom metric (hinge loss) for selecting the best and worst predictions. See this example:

[34]:
```
import numpy as np


def hinge_loss(y_true, y_pred_proba):
    probabilities = np.clip(y_pred_proba.iloc[:, 1], 0.001, 0.999)
    y_true[y_true == 0] = -1

    return np.clip(
        1 - y_true * np.log(probabilities / (1 - probabilities)), a_min=0, a_max=None
    )
```

(continues on next page)

(continued from previous page)

```
report = explain_predictions_best_worst(
    pipeline=pipeline_binary,
    input_features=X,
    y_true=y,
    include_explainer_values=True,
    num_to_explain=5,
    metric=hinge_loss,
)

print(report)
```

```
Random Forest Classifier w/ Label Encoder + Imputer

{'Label Encoder': {'positive_label': None}, 'Imputer': {'categorical_impute_strategy':
↪'most_frequent', 'numeric_impute_strategy': 'mean', 'boolean_impute_strategy': 'most_
↪frequent', 'categorical_fill_value': None, 'numeric_fill_value': None, 'boolean_fill_
↪value': None}, 'Random Forest Classifier': {'n_estimators': 100, 'max_depth': 6, 'n_
↪jobs': -1}}

        Best 1 of 5

                Predicted Probabilities: [benign: 0.03, malignant: 0.97]
                Predicted Value: malignant
                Target Value: malignant
                hinge_loss: 0.0
                Index ID: 0

                    Feature Name       Feature Value   Contribution to Prediction   SHAP␣
↪Value
                ␣
↪==============================================================================
                worst concave points       0.27                    +                  0.
↪08
                  worst perimeter         184.60                   +                  0.
↪08
                mean concave points        0.15                    +                  0.
↪08


        Best 2 of 5

                Predicted Probabilities: [benign: 0.998, malignant: 0.002]
                Predicted Value: benign
                Target Value: benign
                hinge_loss: 0.0
                Index ID: 388

                    Feature Name       Feature Value   Contribution to Prediction   SHAP␣
↪Value
                ␣
↪==============================================================================
                worst concave points       0.08                    -                 -0.
↪05
```

(continues on next page)

(continued from previous page)

```
                mean concave points        0.03                    -                   -0.
↪06
                  worst perimeter          79.73                   -                   -0.
↪07


        Best 3 of 5

                Predicted Probabilities: [benign: 0.988, malignant: 0.012]
                Predicted Value: benign
                Target Value: benign
                hinge_loss: 0.0
                Index ID: 387

                    Feature Name       Feature Value   Contribution to Prediction   SHAP␣
↪Value
                ␣
↪==============================================================================
                  worst perimeter          99.66                   -                   -0.
↪05
                worst concave points       0.05                    -                   -0.
↪05
                mean concave points        0.01                    -                   -0.
↪05


        Best 4 of 5

                Predicted Probabilities: [benign: 1.0, malignant: 0.0]
                Predicted Value: benign
                Target Value: benign
                hinge_loss: 0.0
                Index ID: 386

                   Feature Name       Feature Value   Contribution to Prediction   SHAP␣
↪Value
                ␣
↪===============================================================================
                   worst radius           13.13                   -                   -0.
↪04
                  worst perimeter         87.65                   -                   -0.
↪06
                mean concave points       0.03                    -                   -0.
↪06


        Best 5 of 5

                Predicted Probabilities: [benign: 0.969, malignant: 0.031]
                Predicted Value: benign
                Target Value: benign
                hinge_loss: 0.0
```

(continues on next page)

(continued from previous page)

```
                Index ID: 384

                    Feature Name       Feature Value   Contribution to Prediction   SHAP␣
↪Value
               ␣
↪==============================================================================
                worst concave points       0.09                    -                  -0.
↪04
                  worst perimeter          96.59                   -                  -0.
↪05
                mean concave points        0.03                    -                  -0.
↪06


        Worst 1 of 5

                Predicted Probabilities: [benign: 0.409, malignant: 0.591]
                Predicted Value: malignant
                Target Value: benign
                hinge_loss: 1.369
                Index ID: 128

                    Feature Name       Feature Value   Contribution to Prediction   SHAP␣
↪Value
               ␣
↪==============================================================================
                mean concave points        0.09                    +                   0.
↪10
                worst concave points       0.14                    +                   0.
↪09
                   mean concavity          0.11                    +                   0.
↪08


        Worst 2 of 5

                Predicted Probabilities: [benign: 0.39, malignant: 0.61]
                Predicted Value: malignant
                Target Value: benign
                hinge_loss: 1.446
                Index ID: 421

                   Feature Name       Feature Value   Contribution to Prediction   SHAP␣
↪Value
               ␣
↪===============================================================================
                mean concave points       0.06                    +                   0.
↪08
                  mean concavity          0.14                    +                   0.
↪07
                  worst perimeter        114.10                   +                   0.
↪07
```

(continues on next page)

(continued from previous page)

```
        Worst 3 of 5

                Predicted Probabilities: [benign: 0.343, malignant: 0.657]
                Predicted Value: malignant
                Target Value: benign
                hinge_loss: 1.652
                Index ID: 81

                    Feature Name       Feature Value   Contribution to Prediction   SHAP␣
↪Value
               ␣
↪==============================================================================
                worst concave points       0.17                    ++                  0.
↪15
                mean concave points        0.07                    +                   0.
↪11
                 worst compactness         0.48                    +                   0.
↪07


        Worst 4 of 5

                Predicted Probabilities: [benign: 0.266, malignant: 0.734]
                Predicted Value: malignant
                Target Value: benign
                hinge_loss: 2.016
                Index ID: 363

                 Feature Name     Feature Value   Contribution to Prediction   SHAP Value
                ==============================================================================
                worst perimeter      117.20                  +                   0.13
                 worst radius         18.13                  +                   0.12
                  worst area         1009.00                 +                   0.11


        Worst 5 of 5

                Predicted Probabilities: [benign: 1.0, malignant: 0.0]
                Predicted Value: benign
                Target Value: malignant
                hinge_loss: 7.907
                Index ID: 135

                   Feature Name       Feature Value   Contribution to Prediction   SHAP␣
↪Value
               ␣
↪==============================================================================
                   worst radius           14.49                   -                  -0.
↪05
                  worst perimeter         92.04                   -                  -0.
↪06
```

(continues on next page)

(continued from previous page)

```
                mean concave points       0.03                    -                   -0.
↪06
```

### Changing Output Formats

Instead of getting the prediction explanations as text, you can get the report as a python dictionary or pandas dataframe. All you have to do is pass `output_format="dict"` or `output_format="dataframe"` to either `explain_prediction`, `explain_predictions`, or `explain_predictions_best_worst`.

### Single prediction as a dictionary

[35]:
```
import json

single_prediction_report = explain_predictions(
    pipeline=pipeline_binary,
    input_features=X_holdout,
    indices_to_explain=[3],
    y=y_holdout,
    top_k_features=6,
    include_explainer_values=True,
    output_format="dict",
)
print(json.dumps(single_prediction_report, indent=2))
```

```
{
  "explanations": [
    {
      "explanations": [
        {
          "feature_names": [
            "worst concavity",
            "mean concavity",
            "worst area",
            "worst radius",
            "mean concave points",
            "worst perimeter"
          ],
          "feature_values": [
            0.1791,
            0.038,
            599.5,
            14.04,
            0.034,
            92.8
          ],
          "qualitative_explanation": [
            "-",
```

(continues on next page)

(continued from previous page)


```
            "-",
            "-",
            "-",
            "-",
            "-"
          ],
          "quantitative_explanation": [
            -0.023008481104309524,
            -0.02621982146725469,
            -0.033821592020020774,
            -0.04666659740586632,
            -0.0541511910494414,
            -0.05523688273171911
          ],
          "drill_down": {},
          "class_name": "malignant",
          "expected_value": 0.3711208791208791
        }
      ]
    }
  ]
}
```


### Single prediction as a dataframe

[36]:
```
single_prediction_report = explain_predictions(
    pipeline=pipeline_binary,
    input_features=X_holdout,
    indices_to_explain=[3],
    y=y_holdout,
    top_k_features=6,
    include_explainer_values=True,
    output_format="dataframe",
)
single_prediction_report
```

[36]:
```
         feature_names  feature_values qualitative_explanation  \
0      worst concavity          0.1791                       -
1       mean concavity          0.0380                       -
2           worst area        599.5000                       -
3         worst radius         14.0400                       -
4  mean concave points          0.0340                       -
5      worst perimeter         92.8000                       -

   quantitative_explanation class_name  prediction_number
0                 -0.023008  malignant                  0
1                 -0.026220  malignant                  0
2                 -0.033822  malignant                  0
3                 -0.046667  malignant                  0
4                 -0.054151  malignant                  0
5                 -0.055237  malignant                  0
```

### Best and worst predictions as a dictionary

[37]:
```
report = explain_predictions_best_worst(
    pipeline=pipeline_binary,
    input_features=X,
    y_true=y,
    num_to_explain=1,
    top_k_features=6,
    include_explainer_values=True,
    output_format="dict",
)
print(json.dumps(report, indent=2))
```


```
{
  "explanations": [
    {
      "rank": {
        "prefix": "best",
        "index": 1
      },
      "predicted_values": {
        "probabilities": {
          "benign": 1.0,
          "malignant": 0.0
        },
        "predicted_value": "benign",
        "target_value": "benign",
        "error_name": "Cross Entropy",
        "error_value": 0.0001970443507070075,
        "index_id": 475
      },
      "explanations": [
        {
          "feature_names": [
            "mean concavity",
            "worst area",
            "worst radius",
            "worst concave points",
            "worst perimeter",
            "mean concave points"
          ],
          "feature_values": [
            0.05835,
            605.8,
            14.09,
            0.09783,
            93.22,
            0.03078
          ],
          "qualitative_explanation": [
            "-",
            "-",
            "-",
```


(continues on next page)

(continued from previous page)


```
          "-",
          "-",
          "-"
        ],
        "quantitative_explanation": [
          -0.028481050954786636,
          -0.03050522196002462,
          -0.042922079201003216,
          -0.04429366151003684,
          -0.05486784013962313,
          -0.05639460900233733
        ],
        "drill_down": {},
        "class_name": "malignant",
        "expected_value": 0.3711208791208791
      }
    ]
  },
  {
    "rank": {
      "prefix": "worst",
      "index": 1
    },
    "predicted_values": {
      "probabilities": {
        "benign": 1.0,
        "malignant": 0.0
      },
      "predicted_value": "benign",
      "target_value": "malignant",
      "error_name": "Cross Entropy",
      "error_value": 7.986911819330411,
      "index_id": 135
    },
    "explanations": [
      {
        "feature_names": [
          "mean concavity",
          "worst area",
          "worst concave points",
          "worst radius",
          "worst perimeter",
          "mean concave points"
        ],
        "feature_values": [
          0.04711,
          653.6,
          0.09331,
          14.49,
          92.04,
          0.02704
        ],
```


(continues on next page)

(continued from previous page)

```
          "qualitative_explanation": [
            "-",
            "-",
            "-",
            "-",
            "-",
            "-"
          ],
          "quantitative_explanation": [
            -0.029936744551331215,
            -0.03748357654576422,
            -0.04553126236476177,
            -0.0483274199182721,
            -0.06039220265366764,
            -0.060441902449258976
          ],
          "drill_down": {},
          "class_name": "malignant",
          "expected_value": 0.3711208791208791
        }
      ]
    }
  ]
}
```

### Best and worst predictions as a dataframe

[38]:
```
report = explain_predictions_best_worst(
    pipeline=pipeline_binary,
    input_features=X_holdout,
    y_true=y_holdout,
    num_to_explain=1,
    top_k_features=6,
    include_explainer_values=True,
    output_format="dataframe",
)
report
```

[38]:
```
           feature_names  feature_values qualitative_explanation  \
0         mean concavity         0.05928                       -
1             worst area       552.00000                       -
2   worst concave points         0.08411                       -
3           worst radius        13.57000                       -
4    mean concave points         0.03279                       -
5        worst perimeter        86.67000                       -
6         mean concavity         0.04711                       -
7             worst area       653.60000                       -
8   worst concave points         0.09331                       -
9           worst radius        14.49000                       -
10       worst perimeter        92.04000                       -
11   mean concave points         0.02704                       -
```

(continues on next page)

(continued from previous page)

```
    quantitative_explanation class_name  label_benign_probability  \
0                  -0.029022  malignant                       1.0
1                  -0.034112  malignant                       1.0
2                  -0.046896  malignant                       1.0
3                  -0.046928  malignant                       1.0
4                  -0.052902  malignant                       1.0
5                  -0.064320  malignant                       1.0
6                  -0.029937  malignant                       1.0
7                  -0.037484  malignant                       1.0
8                  -0.045531  malignant                       1.0
9                  -0.048327  malignant                       1.0
10                 -0.060392  malignant                       1.0
11                 -0.060442  malignant                       1.0

    label_malignant_probability predicted_value target_value     error_name  \
0                           0.0          benign       benign  Cross Entropy
1                           0.0          benign       benign  Cross Entropy
2                           0.0          benign       benign  Cross Entropy
3                           0.0          benign       benign  Cross Entropy
4                           0.0          benign       benign  Cross Entropy
5                           0.0          benign       benign  Cross Entropy
6                           0.0          benign    malignant  Cross Entropy
7                           0.0          benign    malignant  Cross Entropy
8                           0.0          benign    malignant  Cross Entropy
9                           0.0          benign    malignant  Cross Entropy
10                          0.0          benign    malignant  Cross Entropy
11                          0.0          benign    malignant  Cross Entropy

    error_value  index_id  rank prefix
0      0.000197       502     1   best
1      0.000197       502     1   best
2      0.000197       502     1   best
3      0.000197       502     1   best
4      0.000197       502     1   best
5      0.000197       502     1   best
6      7.986912       135     1  worst
7      7.986912       135     1  worst
8      7.986912       135     1  worst
9      7.986912       135     1  worst
10     7.986912       135     1  worst
11     7.986912       135     1  worst
```

### 4.6.6 Force Plots

Force plots can be generated to predict single or multiple rows for binary, multiclass and regression problem types. These use the SHAP algorithm. Here's an example of predicting a single row on a binary classification dataset. The force plots show the predictive power of each of the features in making the negative ("Class: 0") prediction and the positive ("Class: 1") prediction.

[39]:
```
import shap

from evalml.model_understanding.force_plots import graph_force_plot

rows_to_explain = [0]  # Should be a list of integer indices of the rows to explain.

results = graph_force_plot(
    pipeline_binary,
    rows_to_explain=rows_to_explain,
    training_data=X_holdout,
    y=y_holdout,
)

for result in results:
    for cls in result:
        print("Class:", cls)
        display(result[cls]["plot"])
```

```
<IPython.core.display.HTML object>
```

```
Class: malignant
```

```
<shap.plots._force.AdditiveForceVisualizer at 0x7fbeda9254c0>
```

Here's an example of a force plot explaining multiple predictions on a multiclass problem. These plots show the force plots for each row arranged as consecutive columns that can be ordered by the dropdown above. Clicking the column indicates which row explanation is underneath.

[40]:
```
rows_to_explain = [
    0,
    1,
    2,
    3,
    4,
]  # Should be a list of integer indices of the rows to explain.

results = graph_force_plot(
    pipeline_multi, rows_to_explain=rows_to_explain, training_data=X_multi, y=y_multi
)

for idx, result in enumerate(results):
    print("Row:", idx)
    for cls in result:
        print("Class:", cls)
        display(result[cls]["plot"])
```

```
<IPython.core.display.HTML object>
```

```
Row: 0
Class: class_0
```

```
<shap.plots._force.AdditiveForceVisualizer at 0x7fbec2247970>
```

```
Class: class_1
```

```
<shap.plots._force.AdditiveForceVisualizer at 0x7fbec21e9dc0>
```

```
Class: class_2
```

```
<shap.plots._force.AdditiveForceVisualizer at 0x7fbec23e1a90>
```

```
Row: 1
Class: class_0
```

```
<shap.plots._force.AdditiveForceVisualizer at 0x7fbec21f4610>
```

```
Class: class_1
```

```
<shap.plots._force.AdditiveForceVisualizer at 0x7fbec21f4fa0>
```

```
Class: class_2
```

```
<shap.plots._force.AdditiveForceVisualizer at 0x7fbec21f4040>
```

```
Row: 2
Class: class_0
```

```
<shap.plots._force.AdditiveForceVisualizer at 0x7fbec21f4250>
```

```
Class: class_1
```

```
<shap.plots._force.AdditiveForceVisualizer at 0x7fbec21f48e0>
```

```
Class: class_2
```

```
<shap.plots._force.AdditiveForceVisualizer at 0x7fbec21f4640>
```

```
Row: 3
Class: class_0
```

```
<shap.plots._force.AdditiveForceVisualizer at 0x7fbec21f46a0>
```

```
Class: class_1
```

```
<shap.plots._force.AdditiveForceVisualizer at 0x7fbec21f4e20>
```

```
Class: class_2
```

```
<shap.plots._force.AdditiveForceVisualizer at 0x7fbec21f4730>
```

```
Row: 4
Class: class_0
```

```
<shap.plots._force.AdditiveForceVisualizer at 0x7fbec21f45e0>
```

```
Class: class_1
```

```
<shap.plots._force.AdditiveForceVisualizer at 0x7fbec21f4be0>
```

```
Class: class_2
```

```
<shap.plots._force.AdditiveForceVisualizer at 0x7fbec21f4130>
```

## 4.7 Data Checks

EvalML provides data checks to help guide you in achieving the highest performing model. These utility functions help deal with problems such as overfitting, abnormal data, and missing data. These data checks can be found under `evalml/data_checks`. Below we will cover examples for each available data check in EvalML, as well as the `DefaultDataChecks` collection of data checks.

### 4.7.1 Missing Data

Missing data or rows with `NaN` values provide many challenges for machine learning pipelines. In the worst case, many algorithms simply will not run with missing data! EvalML pipelines contain imputation *components* to ensure that doesn't happen. Imputation works by approximating missing values with existing values. However, if a column contains a high number of missing values, a large percentage of the column would be approximated by a small percentage. This could potentially create a column without useful information for machine learning pipelines. By using `NullDataCheck`, EvalML will alert you to this potential problem by returning the columns that pass the missing values threshold.

[1]:
```
import numpy as np
import pandas as pd

from evalml.data_checks import NullDataCheck

X = pd.DataFrame(
    [[1, 2, 3], [0, 4, np.nan], [1, 4, np.nan], [9, 4, np.nan], [8, 6, np.nan]]
)

null_check = NullDataCheck(pct_null_col_threshold=0.8, pct_null_row_threshold=0.8)
messages = null_check.validate(X)

errors = [message for message in messages if message["level"] == "error"]
warnings = [message for message in messages if message["level"] == "warning"]

for warning in warnings:
    print("Warning:", warning["message"])

for error in errors:
    print("Error:", error["message"])
```

```
Warning: Column(s) '2' are 80.0% or more null
```

### 4.7.2 Abnormal Data

EvalML provides a few data checks to check for abnormal data:

- `NoVarianceDataCheck`
- `ClassImbalanceDataCheck`
- `TargetLeakageDataCheck`
- `InvalidTargetDataCheck`
- `IDColumnsDataCheck`
- `OutliersDataCheck`

- HighVarianceCVDataCheck
- MulticollinearityDataCheck
- UniquenessDataCheck
- TargetDistributionDataCheck
- DateTimeFormatDataCheck
- TimeSeriesParametersDataCheck
- TimeSeriesSplittingDataCheck

### Zero Variance

Data with zero variance indicates that all values are identical. If a feature has zero variance, it is not likely to be a useful feature. Similarly, if the target has zero variance, there is likely something wrong. NoVarianceDataCheck checks if the target or any feature has only one unique value and alerts you to any such columns.

[2]:
```
from evalml.data_checks import NoVarianceDataCheck

X = pd.DataFrame({"no var col": [0, 0, 0], "good col": [0, 4, 1]})
y = pd.Series([1, 0, 1])
no_variance_data_check = NoVarianceDataCheck()
messages = no_variance_data_check.validate(X, y)

warnings = [message for message in messages if message["level"] == "warning"]

for warning in warnings:
    print("Warning:", warning["message"])
```

```
Warning: 'no var col' has 1 unique value.
```

Note that you can set NaN to count as an unique value, but NoVarianceDataCheck will still return a warning if there is only one unique non-NaN value in a given column.

[3]:
```
from evalml.data_checks import NoVarianceDataCheck

X = pd.DataFrame(
    {
        "no var col": [0, 0, 0],
        "no var col with nan": [1, np.nan, 1],
        "good col": [0, 4, 1],
    }
)
y = pd.Series([1, 0, 1])

no_variance_data_check = NoVarianceDataCheck(count_nan_as_value=True)
messages = no_variance_data_check.validate(X, y)

warnings = [message for message in messages if message["level"] == "warning"]

for warning in warnings:
    print("Warning:", warning["message"])
```

```
Warning: 'no var col' has 1 unique value.
Warning: 'no var col with nan' has two unique values including nulls. Consider encoding␣
↪the nulls for this column to be useful for machine learning.
```

### Class Imbalance

For classification problems, the distribution of examples across each class can vary. For small variations, this is normal and expected. However, when the number of examples for each class label is disproportionately biased or skewed towards a particular class (or classes), it can be difficult for machine learning models to predict well. In addition, having a low number of examples for a given class could mean that one or more of the CV folds generated for the training data could only have few or no examples from that class. This may cause the model to only predict the majority class and ultimately resulting in a poor-performant model.

`ClassImbalanceDataCheck` checks if the target labels are imbalanced beyond a specified threshold for a certain number of CV folds. It returns `DataCheckError` messages for any classes that have less samples than double the number of CV folds specified (since that indicates the likelihood of having at little to no samples of that class in a given fold), and `DataCheckWarning` messages for any classes that fall below the set threshold percentage.

[4]:
```
from evalml.data_checks import ClassImbalanceDataCheck

X = pd.DataFrame([[1, 2, 0, 1], [4, 1, 9, 0], [4, 4, 8, 3], [9, 2, 7, 1]])
y = pd.Series([0, 1, 1, 1, 1])

class_imbalance_check = ClassImbalanceDataCheck(threshold=0.25, num_cv_folds=4)
messages = class_imbalance_check.validate(X, y)

errors = [message for message in messages if message["level"] == "error"]
warnings = [message for message in messages if message["level"] == "warning"]

for warning in warnings:
    print("Warning:", warning["message"])

for error in errors:
    print("Error:", error["message"])
```

```
Warning: The following labels fall below 25% of the target: [0]
Warning: The following labels in the target have severe class imbalance because they␣
↪fall under 25% of the target and have less than 100 samples: [0]
Error: The number of instances of these targets is less than 2 * the number of cross␣
↪folds = 8 instances: [0, 1]
```

### Target Leakage

Target leakage, also known as data leakage, can occur when you train your model on a dataset that includes information that should not be available at the time of prediction. This causes the model to score suspiciously well, but perform poorly in production. `TargetLeakageDataCheck` checks for features that could potentially be "leaking" information by calculating the Pearson correlation coefficient between each feature and the target to warn users if there are features are highly correlated with the target. Currently, only numerical features are considered.

[5]:
```
from evalml.data_checks import TargetLeakageDataCheck

X = pd.DataFrame(
```

(continues on next page)

(continued from previous page)

```
    {"leak": [10, 42, 31, 51, 61], "x": [42, 54, 12, 64, 12], "y": [12, 5, 13, 74, 24]}
)
y = pd.Series([10, 42, 31, 51, 40])

target_leakage_check = TargetLeakageDataCheck(pct_corr_threshold=0.8)
messages = target_leakage_check.validate(X, y)

errors = [message for message in messages if message["level"] == "error"]
warnings = [message for message in messages if message["level"] == "warning"]

for warning in warnings:
    print("Warning:", warning["message"])

for error in errors:
    print("Error:", error["message"])
```

### Invalid Target Data

The `InvalidTargetDataCheck` checks if the target data contains any missing or invalid values. Specifically:

- if any of the target values are missing, a `DataCheckError` message is returned
- if the specified problem type is a binary classification problem but there is more or less than two unique values in the target, a `DataCheckError` message is returned
- if binary classification target classes are numeric values not equal to {0, 1}, a `DataCheckError` message is returned because it can cause unpredictable behavior when passed to pipelines

[6]:
```
from evalml.data_checks import InvalidTargetDataCheck

X = pd.DataFrame({})
y = pd.Series([0, 1, None, None])

invalid_target_check = InvalidTargetDataCheck("binary", "Log Loss Binary")
messages = invalid_target_check.validate(X, y)

errors = [message for message in messages if message["level"] == "error"]
warnings = [message for message in messages if message["level"] == "warning"]

for warning in warnings:
    print("Warning:", warning["message"])

for error in errors:
    print("Error:", error["message"])
```

```
Warning: Input target and features have different lengths
Warning: Input target and features have mismatched indices. Details will include the␣
↪first 10 mismatched indices.
Error: 2 row(s) (50.0%) of target values are null
```

### ID Columns

ID columns in your dataset provide little to no benefit to a machine learning pipeline as the pipeline cannot extrapolate useful information from unique identifiers. Thus, `IDColumnsDataCheck` reminds you if these columns exists. In the given example, 'user_number' and 'revenue_id' columns are both identified as potentially being unique identifiers that should be removed.

[7]:
```
from evalml.data_checks import IDColumnsDataCheck

X = pd.DataFrame(
    [[0, 53, 6325, 5], [1, 90, 6325, 10], [2, 90, 18, 20]],
    columns=["user_number", "cost", "revenue", "revenue_id"],
)

id_col_check = IDColumnsDataCheck(id_threshold=0.9)
messages = id_col_check.validate(X)

errors = [message for message in messages if message["level"] == "error"]
warnings = [message for message in messages if message["level"] == "warning"]

for warning in warnings:
    print("Warning:", warning["message"])

for error in errors:
    print("Error:", error["message"])
```

```
Warning: Columns 'user_number', 'revenue_id' are 90.0% or more likely to be an ID column
```

Primary key columns however, can be useful. Primary key columns are typically the first column in the dataset, have all unique values, and are either named `ID` or a name that ends with `_id`. Though they are ignored from the modeling process, they can be used as an identifier to query on before or after the modeling process. `IDColumnsDataCheck` will also remind you if it finds that the first column of the DataFrame is a primary key. In the given example, `user_id` is identified as a primary key, while `revenue_id` was identified as a regular unique identifier.

[8]:
```
from evalml.data_checks import IDColumnsDataCheck

X = pd.DataFrame(
    [[0, 53, 6325, 5], [1, 90, 6325, 10], [2, 90, 18, 20]],
    columns=["user_id", "cost", "revenue", "revenue_id"],
)

id_col_check = IDColumnsDataCheck(id_threshold=0.9)
messages = id_col_check.validate(X)

errors = [message for message in messages if message["level"] == "error"]
warnings = [message for message in messages if message["level"] == "warning"]

for warning in warnings:
    print("Warning:", warning["message"])

for error in errors:
    print("Error:", error["message"])
```

```
Warning: The first column 'user_id' is likely to be the primary key
Warning: Columns 'revenue_id' are 90.0% or more likely to be an ID column
```

### Multicollinearity

The `MulticollinearityDataCheck` data check is used in to detect if are any set of features that are likely to be multicollinear. Multicollinear features affect the performance of a model, but more importantly, it may greatly impact model interpretation. EvalML uses mutual information to determine collinearity.

[9]:
```
from evalml.data_checks import MulticollinearityDataCheck

y = pd.Series([1, 0, 2, 3, 4])
X = pd.DataFrame(
    {
        "col_1": y,
        "col_2": y * 3,
        "col_3": ~y,
        "col_4": y / 2,
        "col_5": y + 1,
        "not_collinear": [0, 1, 0, 0, 0],
    }
)

multi_check = MulticollinearityDataCheck(threshold=0.95)
messages = multi_check.validate(X)

errors = [message for message in messages if message["level"] == "error"]
warnings = [message for message in messages if message["level"] == "warning"]

for warning in warnings:
    print("Warning:", warning["message"])

for error in errors:
    print("Error:", error["message"])
```

### Uniqueness

The `UniquenessDataCheck` is used to detect columns with either too unique or not unique enough values. For regression type problems, the data is checked for a lower limit of uniqueness. For multiclass type problems, the data is checked for an upper limit.

[10]:
```
import pandas as pd
from evalml.data_checks import UniquenessDataCheck

X = pd.DataFrame(
    {
        "most_unique": [float(x) for x in range(10)],  # [0,1,2,3,4,5,6,7,8,9]
        "more_unique": [x % 5 for x in range(10)],  # [0,1,2,3,4,0,1,2,3,4]
        "unique": [x % 3 for x in range(10)],  # [0,1,2,0,1,2,0,1,2,0]
        "less_unique": [x % 2 for x in range(10)],  # [0,1,0,1,0,1,0,1,0,1]
        "not_unique": [float(1) for x in range(10)],
    }
)  # [1,1,1,1,1,1,1,1,1,1]

uniqueness_check = UniquenessDataCheck(problem_type="regression", threshold=0.5)
```

(continues on next page)

(continued from previous page)

```
messages = uniqueness_check.validate(X)

errors = [message for message in messages if message["level"] == "error"]
warnings = [message for message in messages if message["level"] == "warning"]

for warning in warnings:
    print("Warning:", warning["message"])

for error in errors:
    print("Error:", error["message"])
```

```
Warning: Input columns 'not_unique' for regression problem type are not unique enough.
```

### Sparsity

The `SparsityDataCheck` is used to identify features that contain a sparsity of values.

[11]:
```
from evalml.data_checks import SparsityDataCheck

X = pd.DataFrame(
    {
        "most_sparse": [float(x) for x in range(10)],  # [0,1,2,3,4,5,6,7,8,9]
        "more_sparse": [x % 5 for x in range(10)],  # [0,1,2,3,4,0,1,2,3,4]
        "sparse": [x % 3 for x in range(10)],  # [0,1,2,0,1,2,0,1,2,0]
        "less_sparse": [x % 2 for x in range(10)],  # [0,1,0,1,0,1,0,1,0,1]
        "not_sparse": [float(1) for x in range(10)],
    }
)  # [1,1,1,1,1,1,1,1,1,1]


sparsity_check = SparsityDataCheck(
    problem_type="multiclass", threshold=0.4, unique_count_threshold=3
)
messages = sparsity_check.validate(X)

errors = [message for message in messages if message["level"] == "error"]
warnings = [message for message in messages if message["level"] == "warning"]


for warning in warnings:
    print("Warning:", warning["message"])

for error in errors:
    print("Error:", error["message"])
```

```
Warning: Input columns ('most_sparse', 'more_sparse', 'sparse') for multiclass problem␣
↪type are too sparse.
```

### Outliers

Outliers are observations that differ significantly from other observations in the same sample. Many machine learning pipelines suffer in performance if outliers are not dropped from the training set as they are not representative of the data. `OutliersDataCheck()` uses IQR to notify you if a sample can be considered an outlier.

Below we generate a random dataset with some outliers.

[12]:
```
data = np.tile(np.arange(10) * 0.01, (100, 10))
X = pd.DataFrame(data=data)

# generate some outliers in columns 3, 25, 55, and 72
X.iloc[0, 3] = -10000
X.iloc[3, 25] = 10000
X.iloc[5, 55] = 10000
X.iloc[10, 72] = -10000
```

We then utilize `OutliersDataCheck()` to rediscover these outliers.

[13]:
```
from evalml.data_checks import OutliersDataCheck

outliers_check = OutliersDataCheck()
messages = outliers_check.validate(X)

errors = [message for message in messages if message["level"] == "error"]
warnings = [message for message in messages if message["level"] == "warning"]

for warning in warnings:
    print("Warning:", warning["message"])

for error in errors:
    print("Error:", error["message"])
```

```
Warning: Column(s) '3', '25', '55', '72' are likely to have outlier data.
```

### Target Distribution

Target data can come in a variety of distributions, such as Gaussian or Lognormal. When we work with machine learning models, we feed data into an estimator that learns from the training data provided. Sometimes the data can be significantly spread out with a long tail or outliers, which could lead to a lognormal distribution. This can cause machine learning model performance to suffer.

To help the estimators better understand the underlying relationships in the data between the features and the target, we can use the `TargetDistributionDataCheck` to identify such a distribution.

[14]:
```
from scipy.stats import lognorm
from evalml.data_checks import TargetDistributionDataCheck

data = np.tile(np.arange(10) * 0.01, (100, 10))
X = pd.DataFrame(data=data)
y = pd.Series(lognorm.rvs(s=0.4, loc=1, scale=1, size=100))

target_dist_check = TargetDistributionDataCheck()
messages = target_dist_check.validate(X, y)
```

(continues on next page)

(continued from previous page)

```
errors = [message for message in messages if message["level"] == "error"]
warnings = [message for message in messages if message["level"] == "warning"]

for warning in warnings:
    print("Warning:", warning["message"])

for error in errors:
    print("Error:", error["message"])
```

```
Warning: Target may have a lognormal distribution.
```

### Datetime Format

Datetime information is a necessary component of time series problems, but sometimes the data we deal with may contain flaws that make it impossible for time series models to work with them. For example, in order to identify a frequency in the datetime information there has to be equal interval spacing between data points i.e. January 1, 2021, January 3, 2021, January 5, 2021, ...etc which are separated by two days. If instead there are random jumps in the datetime data i.e. January 1, 2021, January 3, 2021, January 12, 2021, then a frequency can't be inferred. Another common issue with time series models are that they can't handle datetime information that isn't properly sorted. Datetime values that aren't monotonically increasing (sorted in ascending order) will encounter this issue and their frequency cannot be inferred.

To make it easy to verify that the datetime column you're working with is properly spaced and sorted, we can leverage the `DatetimeFormatDataCheck`. When initializing the data check, pass in the name of the column that contains your datetime information (or pass in "index" if it's found in either your X or y indices).

[15]:
```
from evalml.data_checks import DateTimeFormatDataCheck

X = pd.DataFrame(
    pd.date_range("January 1, 2021", periods=8, freq="2D"), columns=["dates"]
)
y = pd.Series([1, 2, 4, 2, 1, 2, 3, 1])

# Replaces the last entry with January 16th instead of January 15th
# so that the data is no longer evenly spaced.
X.iloc[7] = "January 16, 2021"

datetime_format_check = DateTimeFormatDataCheck(datetime_column="dates")
messages = datetime_format_check.validate(X, y)

errors = [message for message in messages if message["level"] == "error"]
warnings = [message for message in messages if message["level"] == "warning"]

for warning in warnings:
    print("Warning:", warning["message"])

for error in errors:
    print("Error:", error["message"])

print("--------------------------------")
```

(continues on next page)

(continued from previous page)

```
# Reverses the order of the index datetime values to be decreasing.
X = X[::-1]
messages = datetime_format_check.validate(X, y)

errors = [message for message in messages if message["level"] == "error"]
warnings = [message for message in messages if message["level"] == "warning"]

for warning in warnings:
    print("Warning:", warning["message"])

for error in errors:
    print("Error:", error["message"])
```

```
Error: Column 'dates' has datetime values that do not align with the inferred frequency.
Error: A frequency was detected in column 'dates', but there are faulty datetime values␣
↪that need to be addressed.
--------------------------------
Error: Datetime values must be sorted in ascending order.
Error: No frequency could be detected in column 'dates', possibly due to uneven␣
↪intervals or too many duplicate/missing values.
```

### Time Series Parameters

In order to support time series problem types in AutoML, certain conditions have to be met. - The parameters `gap`, `max_delay`, `forecast_horizon`, and `time_index` have to be passed in to `problem_configuration`. - The values of `gap`, `max_delay`, `forecast_horizon` have to be appropriate for the size of the data.

For point 2 above, this means that the window size (as defined by `gap` + `max_delay` + `forecast_horizon`) has to be less than the number of observations in the data divided by the number of splits + 1. For example, with 100 observations and 3 splits, the split size would be 25. This means that the window size has to be less than 25.

[16]:
```
from evalml.data_checks import TimeSeriesParametersDataCheck

X = pd.DataFrame(pd.date_range("1/1/21", periods=100), columns=["dates"])
y = pd.Series([i % 2 for i in range(100)])

problem_config = {
    "gap": 1,
    "max_delay": 23,
    "forecast_horizon": 1,
    "time_index": "dates",
}

# With 3 splits, the split size will be 25 (100/3+1)
# Since gap + max_delay + forecast_horizon is 25, this will
# throw an error for window size.
ts_params_data_check = TimeSeriesParametersDataCheck(
    problem_configuration=problem_config, n_splits=3
)
messages = ts_params_data_check.validate(X, y)

errors = [message for message in messages if message["level"] == "error"]
```

(continues on next page)

(continued from previous page)

```
warnings = [message for message in messages if message["level"] == "warning"]

for warning in warnings:
    print("Warning:", warning["message"])

for error in errors:
    print("Error:", error["message"])
```

### Time Series Splitting

Due to the nature of time series data, splitting cannot involve shuffling and has to be done in a sequential manner. This means splitting the data into `n_splits` + 1 different sections and increasing the size of the training data by the split size every iteration while keeping the test size equal to the split size.

For every split in the data, the training and validation segments must contain target data that has an example of every class found in the entire target set for time series binary and time series multiclass problems. The reason for this is that many classification machine learning models run into issues if they're trained on data that doesn't contain an instance of a class but then the model is expected to be able to predict for it. For example, with 3 splits and a split size of 25, this means that every training/validation split: (0:25)/(25:50), (0:50)/(50:75), (0:75)/(75:100) must contain at least one instance of all unique target classes in the training and validation set. - At least one instance of both classes in a time series binary problem. - At least one instance of all classes in a time series multiclass problem.

[17]:
```
from evalml.data_checks import TimeSeriesSplittingDataCheck

X = None
y = pd.Series([0 if i < 50 else i % 2 for i in range(100)])

ts_splitting_check = TimeSeriesSplittingDataCheck("time series binary", 3)
messages = ts_splitting_check.validate(X, y)

errors = [message for message in messages if message["level"] == "error"]
warnings = [message for message in messages if message["level"] == "warning"]

for warning in warnings:
    print("Warning:", warning["message"])

for error in errors:
    print("Error:", error["message"])
```

```
Error: Time Series Binary and Time Series Multiclass problem types require every␣
↪training and validation split to have at least one instance of all the target classes.␣
↪The following splits are invalid: [1, 2]
```

### 4.7.3 Data Check Messages

Each data check's `validate` method returns a list of `DataCheckMessage` objects indicating warnings or errors found; warnings are stored as a `DataCheckWarning` *object* and errors are stored as a `DataCheckError` *object*. You can filter the messages returned by a data check by checking for the type of message returned. Below, `NoVarianceDataCheck` returns a list containing a `DataCheckWarning` and a `DataCheckError` message. We can determine which is which by checking the type of each message.

[18]:
```
from evalml.data_checks import NoVarianceDataCheck, DataCheckWarning

X = pd.DataFrame(
    {
        "no var col": [0, 0, 0],
        "no var col with nan": [1, np.nan, 1],
        "good col": [0, 4, 1],
    }
)
y = pd.Series([1, 0, 1])

no_variance_data_check = NoVarianceDataCheck(count_nan_as_value=True)
messages = no_variance_data_check.validate(X, y)

warnings = [message for message in messages if message["level"] == "warning"]

for warning in warnings:
    print("Warning:", warning["message"])
```

```
Warning: 'no var col' has 1 unique value.
Warning: 'no var col with nan' has two unique values including nulls. Consider encoding␣
↪the nulls for this column to be useful for machine learning.
```

### 4.7.4 Writing Your Own Data Check

If you would prefer to write your own data check, you can do so by extending the `DataCheck` class and implementing the `validate(self, X, y)` class method. Below, we've created a new `DataCheck`, `ZeroVarianceDataCheck`, which is similar to `NoVarianceDataCheck` defined in `EvalML`. The `validate(self, X, y)` method should return a dictionary with 'warnings' and 'errors' as keys mapping to list of warnings and errors, respectively.

[19]:
```
from evalml.data_checks import DataCheck


class ZeroVarianceDataCheck(DataCheck):
    def validate(self, X, y):
        messages = []
        if not isinstance(X, pd.DataFrame):
            X = pd.DataFrame(X)
        warning_msg = "Column '{}' has zero variance"
        messages.extend(
            [
                DataCheckError(warning_msg.format(column), self.name)
                for column in X.columns
                if len(X[column].unique()) == 1
            ]
```

(continues on next page)

(continued from previous page)

```
        )
        return messages
```

### 4.7.5 Defining Collections of Data Checks

For convenience, EvalML provides a `DataChecks` class to represent a collection of data checks. We will go over `DefaultDataChecks` (*API reference*), a collection defined to check for some of the most common data issues.

#### Default Data Checks

`DefaultDataChecks` is a collection of data checks defined to check for some of the most common data issues. They include:

- `NullDataCheck`
- `IDColumnsDataCheck`
- `TargetLeakageDataCheck`
- `InvalidTargetDataCheck`
- `TargetDistributionDataCheck` (for regression problem types)
- `ClassImbalanceDataCheck` (for classification problem types)
- `NoVarianceDataCheck`
- `DateTimeFormatDataCheck` (for time series problem types)
- `TimeSeriesParametersDataCheck` (for time series problem types)
- `TimeSeriesSplittingDataCheck` (for time series classification problem types)

### 4.7.6 Writing Your Own Collection of Data Checks

If you would prefer to create your own collection of data checks, you could either write your own data checks class by extending the `DataChecks` class and setting the `self.data_checks` attribute to the list of `DataCheck` classes or objects, or you could pass that list of data checks to the constructor of the `DataChecks` class. Below, we create two identical collections of data checks using the two different methods.

[20]:
```
# Create a subclass of `DataChecks`
from evalml.data_checks import (
    DataChecks,
    NullDataCheck,
    InvalidTargetDataCheck,
    NoVarianceDataCheck,
    ClassImbalanceDataCheck,
    TargetLeakageDataCheck,
)
from evalml.problem_types import ProblemTypes, handle_problem_types


class MyCustomDataChecks(DataChecks):
    data_checks = [
```

(continues on next page)

(continued from previous page)

```
        NullDataCheck,
        InvalidTargetDataCheck,
        NoVarianceDataCheck,
        TargetLeakageDataCheck,
    ]

    def __init__(self, problem_type, objective):
        """
        A collection of basic data checks.
        Args:
            problem_type (str): The problem type that is being validated. Can be␣
↪regression, binary, or multiclass.
        """
        if handle_problem_types(problem_type) == ProblemTypes.REGRESSION:
            super().__init__(
                self.data_checks,
                data_check_params={
                    "InvalidTargetDataCheck": {
                        "problem_type": problem_type,
                        "objective": objective,
                    }
                },
            )
        else:
            super().__init__(
                self.data_checks + [ClassImbalanceDataCheck],
                data_check_params={
                    "InvalidTargetDataCheck": {
                        "problem_type": problem_type,
                        "objective": objective,
                    }
                },
            )


custom_data_checks = MyCustomDataChecks(
    problem_type=ProblemTypes.REGRESSION, objective="R2"
)
for data_check in custom_data_checks.data_checks:
    print(data_check.name)
```

```
NullDataCheck
InvalidTargetDataCheck
NoVarianceDataCheck
TargetLeakageDataCheck
```

[21]:
```
# Pass list of data checks to the `data_checks` parameter of DataChecks
same_custom_data_checks = DataChecks(
    data_checks=[
        NullDataCheck,
        InvalidTargetDataCheck,
        NoVarianceDataCheck,
```

(continues on next page)

(continued from previous page)

```
        TargetLeakageDataCheck,
    ],
    data_check_params={
        "InvalidTargetDataCheck": {
            "problem_type": ProblemTypes.REGRESSION,
            "objective": "R2",
        }
    },
)
for data_check in custom_data_checks.data_checks:
    print(data_check.name)
```

```
NullDataCheck
InvalidTargetDataCheck
NoVarianceDataCheck
TargetLeakageDataCheck
```

## 4.8 Understanding Data Check Actions

EvalML streamlines the creation and implementation of machine learning models for tabular data. One of the many features it offers is data checks, which help determine the health of our data before we train a model on it. These data checks have associated actions with them and will be shown in this notebook. In our default data checks, we have the following checks:

- `NullDataCheck`: Checks whether the rows or columns are null or highly null
- `IDColumnsDataCheck`: Checks for columns that could be ID columns
- `TargetLeakageDataCheck`: Checks if any of the input features have high association with the targets
- `InvalidTargetDataCheck`: Checks if there are null or other invalid values in the target
- `NoVarianceDataCheck`: Checks if either the target or any features have no variance

EvalML has additional data checks that can be seen here, with usage examples here. Below, we will walk through usage of EvalML's default data checks and actions.

First, we import the necessary requirements to demonstrate these checks.

[1]:
```
import woodwork as ww
import pandas as pd
from evalml import AutoMLSearch
from evalml.demos import load_fraud
from evalml.preprocessing import split_data
```

Let's look at the input feature data. EvalML uses the Woodwork library to represent this data. The demo data that EvalML returns is a Woodwork DataTable and DataColumn.

[2]:
```
X, y = load_fraud(n_rows=1500)
X.head()
```

```
             Number of Features
Boolean                       1
Categorical                   6
Numeric                       5
```

(continues on next page)

(continued from previous page)

```
Number of training examples: 1500
Targets
False    86.60%
True     13.40%
Name: fraud, dtype: object
```

[2]:
```
    card_id  store_id            datetime  amount currency  customer_present  \
id
0     32261      8516 2019-01-01 00:12:26   24900      CUC              True
1     16434      8516 2019-01-01 09:42:03   15789      MYR             False
2     23468      8516 2019-04-17 08:17:01    1883      AUD             False
3     14364      8516 2019-01-30 11:54:30   82120      KRW              True
4     29407      8516 2019-05-01 17:59:36   25745      MUR              True

   expiration_date          provider       lat       lng            region  \
id
0            08/24        Mastercard  38.58894 -89.99038  Fairview Heights
1            11/21          Discover  38.58894 -89.99038  Fairview Heights
2            09/27          Discover  38.58894 -89.99038  Fairview Heights
3            09/20      JCB 16 digit  38.58894 -89.99038  Fairview Heights
4            09/22  American Express  38.58894 -89.99038  Fairview Heights

   country
id
0       US
1       US
2       US
3       US
4       US
```

### 4.8.1 Adding noise and unclean data

This data is already clean and compatible with EvalML's `AutoMLSearch`. In order to demonstrate EvalML default data checks, we will add the following:

- A column of mostly null values (<0.5% non-null)
- A column with low/no variance
- A row of null values
- A missing target value

We will add the first two columns to the whole dataset and we will only add the last two to the training data. Note: these only represent some of the scenarios that EvalML default data checks can catch.

[3]:
```
# add a column with no variance in the data
X["no_variance"] = [1 for _ in range(X.shape[0])]

# add a column with >99.5% null values
X["mostly_nulls"] = [None] * (X.shape[0] - 5) + [i for i in range(5)]

# since we changed the data, let's reinitialize the woodwork datatable
```

(continues on next page)

(continued from previous page)

```
X.ww.init()
# let's split some training and validation data
X_train, X_valid, y_train, y_valid = split_data(X, y, problem_type="binary")
```

[4]:
```
# make row 1 all nan values
X_train.iloc[1] = [None] * X_train.shape[1]

# make one of the target values null
y_train[990] = None

X_train.ww.init()
y_train = ww.init_series(y_train, logical_type="Categorical")
# Let's take another look at the new X_train data
X_train
```

[4]:
```
      card_id  store_id            datetime  amount currency  \
id
872     15492      2868 2019-08-03 02:50:04   80719      HNL
1477     <NA>      <NA>                 NaT    <NA>      NaN
158     22440      6813 2019-07-12 11:07:25    1849      SEK
808      8096      8096 2019-06-11 21:33:36   41358      MOP
336     33270      1529 2019-03-23 21:44:00   32594      CUC
...       ...       ...                 ...     ...      ...
339      8484      5358 2019-01-10 07:47:28   89503      GMD
1383    17565      3929 2019-01-15 01:11:02   14264      DKK
893       108        44 2019-05-17 00:53:39   93218      SLL
385     29983       152 2019-06-09 06:50:29   41105      RWF
1074    26197      4927 2019-05-22 15:57:27   50481      MNT

      customer_present expiration_date          provider       lat        lng  \
id
872               True           08/27  American Express   5.47090  100.24529
1477              <NA>             NaN               NaN       NaN        NaN
158               True           09/20  American Express  26.26490   81.54855
808               True           04/29     VISA 13 digit  59.37722   28.19028
336              False           04/22        Mastercard  51.39323    0.47713
...                ...             ...               ...       ...        ...
339              False           11/24           Maestro  47.30997    8.52462
1383              True           06/20     VISA 13 digit  50.72043   11.34046
893               True           12/24      JCB 16 digit  15.72892  120.57224
385              False           07/20      JCB 16 digit  -6.80000   39.25000
1074             False           05/26      JCB 15 digit  41.00510  -73.78458

             region country  no_variance  mostly_nulls
id
872   Batu Feringgi      MY            1          <NA>
1477            NaN     NaN         <NA>          <NA>
158            Jais      IN            1          <NA>
808           Narva      EE            1          <NA>
336          Strood      GB            1          <NA>
...             ...     ...          ...           ...
339        Adliswil      CH            1          <NA>
```

(continues on next page)

(continued from previous page)

```
1383     Rudolstadt      DE            1          <NA>
893          Burgos      PH            1          <NA>
385        Magomeni      TZ            1          <NA>
1074      Scarsdale      US            1          <NA>

[1200 rows x 14 columns]
```

If we call `AutoMLSearch.search()` on this data, the search will fail due to the columns and issues we've added above. Note: we use a try/except here to catch the resulting ValueError that AutoMLSearch raises.

[5]:
```
automl = AutoMLSearch(X_train=X_train, y_train=y_train, problem_type="binary")
try:
    automl.search()
except ValueError as e:
    # to make the error message more distinct
    print("=" * 80, "\n")
    print("Search errored out! Message received is: {}".format(e))
    print("=" * 80, "\n")
```

```
================================================================================

Search errored out! Message received is: Input y contains NaN.
================================================================================
```

We can use the `search_iterative()` function provided in EvalML to determine what potential health issues our data has. We can see that this search_iterative function is a public method available through `evalml.automl` and is different from the search function of the `AutoMLSearch` class in EvalML. This `search_iterative()` function allows us to run the default data checks on the data, and, if there are no errors, automatically runs `AutoMLSearch.search()`.

[6]:
```
from evalml.automl import search_iterative

automl, messages = search_iterative(X_train, y_train, problem_type="binary")
automl, messages
```

[6]:
```
(None,
 [{'message': '1 out of 1200 rows are 95.0% or more null',
   'data_check_name': 'NullDataCheck',
   'level': 'warning',
   'details': {'columns': None,
    'rows': [1477],
    'pct_null_cols': id
    1477    1.0
    dtype: float64},
   'code': 'HIGHLY_NULL_ROWS',
   'action_options': [{'code': 'DROP_ROWS',
     'data_check_name': 'NullDataCheck',
     'metadata': {'columns': None, 'rows': [1477]},
     'parameters': {}}]},
  {'message': "Column(s) 'mostly_nulls' are 95.0% or more null",
   'data_check_name': 'NullDataCheck',
   'level': 'warning',
   'details': {'columns': ['mostly_nulls'],
```

(continues on next page)

(continued from previous page)

```
    'rows': None,
    'pct_null_rows': {'mostly_nulls': 0.9966666666666667}},
   'code': 'HIGHLY_NULL_COLS',
   'action_options': [{'code': 'DROP_COL',
     'data_check_name': 'NullDataCheck',
     'metadata': {'columns': ['mostly_nulls'], 'rows': None},
     'parameters': {}}]},
  {'message': '1 row(s) (0.08333333333333334%) of target values are null',
   'data_check_name': 'InvalidTargetDataCheck',
   'level': 'error',
   'details': {'columns': None,
    'rows': [990],
    'num_null_rows': 1,
    'pct_null_rows': 0.08333333333333334},
   'code': 'TARGET_HAS_NULL',
   'action_options': [{'code': 'DROP_ROWS',
     'data_check_name': 'InvalidTargetDataCheck',
     'metadata': {'columns': None, 'rows': [990], 'is_target': True},
     'parameters': {}}]},
  {'message': "'no_variance' has 1 unique value.",
   'data_check_name': 'NoVarianceDataCheck',
   'level': 'warning',
   'details': {'columns': ['no_variance'], 'rows': None},
   'code': 'NO_VARIANCE',
   'action_options': [{'code': 'DROP_COL',
     'data_check_name': 'NoVarianceDataCheck',
     'metadata': {'columns': ['no_variance'], 'rows': None},
     'parameters': {}}]}])
```

The return value of the `search_iterative` function above is a tuple. The first element is the `AutoMLSearch` object if it runs (and `None` otherwise), and the second element is a dictionary of potential warnings and errors that the default data checks find on the passed-in `X` and `y` data. In this dictionary, warnings are suggestions that the data checks give that can useful to address to make the search better but will not break `AutoMLSearch`. On the flip side, errors indicate issues that will break `AutoMLSearch` and need to be addressed by the user.

Above, we can see that there were errors so search did not automatically run.

### 4.8.2 Addressing warnings and errors

We can automatically address the warnings and errors returned by `search_iterative` by using `make_pipeline_from_data_check_output`, a utility method that creates a pipeline that will automatically clean up our data. We just need to pass this method the messages from running `DataCheck.validate()` and our problem type.

[7]:
```
from evalml.pipelines.utils import make_pipeline_from_data_check_output

actions_pipeline = make_pipeline_from_data_check_output("binary", messages)
actions_pipeline.fit(X_train, y_train)
X_train_cleaned, y_train_cleaned = actions_pipeline.transform(X_train, y_train)
print(
    "The new length of X_train is {} and y_train is {}".format(
        len(X_train_cleaned), len(X_train_cleaned)
```

(continues on next page)

(continued from previous page)

```
    )
)
```

```
The new length of X_train is 1198 and y_train is 1198
```

Now, we can run `search_iterative` to completion.

[8]:
```
results_cleaned = search_iterative(
    X_train_cleaned, y_train_cleaned, problem_type="binary"
)
```

Note that this time, we get an `AutoMLSearch` object returned to us as the first element of the tuple. We can use and inspect the `AutoMLSearch` object as needed.

[9]:
```
automl_object = results_cleaned[0]
automl_object.rankings
```

[9]:
```
   id                                     pipeline_name  search_order  \
0   1  Random Forest Classifier w/ Label Encoder + Da...             1
1   0       Mode Baseline Binary Classification Pipeline             0

   ranking_score  mean_cv_score  standard_deviation_cv_score  \
0       0.262989       0.262989                     0.007899
1       4.843912       4.843912                     0.049015

   percent_better_than_baseline  high_variance_cv  \
0                     94.570726             False
1                      0.000000             False

                                          parameters
0  {'Label Encoder': {'positive_label': None}, 'D...
1  {'Label Encoder': {'positive_label': None}, 'B...
```

If we check the second element in the tuple, we can see that there are no longer any warnings or errors detected!

[10]:
```
data_check_results = results_cleaned[1]
data_check_results
```

[10]:
```
[]
```

### 4.8.3 Only addressing DataCheck errors

Previously, we used `make_pipeline_from_actions` to address all of the warnings and errors returned by `search_iterative`. We will now show how we can also manually address errors to allow AutoMLSearch to run, and how ignoring warnings will come at the expense of performance.

We can print out the errors first to make it easier to read, and then we'll create new features and targets from the original training data.

[11]:
```
errors = [message for message in messages if message["level"] == "error"]
errors
```

[11]:
```
[{'message': '1 row(s) (0.08333333333333334%) of target values are null',
  'data_check_name': 'InvalidTargetDataCheck',
```

(continues on next page)

(continued from previous page)

```
  'level': 'error',
  'details': {'columns': None,
   'rows': [990],
   'num_null_rows': 1,
   'pct_null_rows': 0.08333333333333334},
  'code': 'TARGET_HAS_NULL',
  'action_options': [{'code': 'DROP_ROWS',
    'data_check_name': 'InvalidTargetDataCheck',
    'metadata': {'columns': None, 'rows': [990], 'is_target': True},
    'parameters': {}}]}]
```

[12]:
```
# copy the DataTables to new variables
X_train_no_errors = X_train.copy()
y_train_no_errors = y_train.copy()

# We address the errors by looking at the resulting dictionary errors listed

# let's address the `TARGET_HAS_NULL` error
y_train_no_errors.fillna(False, inplace=True)

# let's reinitialize the Woodwork DataTable
X_train_no_errors.ww.init()
X_train_no_errors.head()
```

[12]:
```
      card_id  store_id            datetime  amount currency  \
id
872     15492      2868 2019-08-03 02:50:04   80719      HNL
1477     <NA>      <NA>                 NaT    <NA>      NaN
158     22440      6813 2019-07-12 11:07:25    1849      SEK
808      8096      8096 2019-06-11 21:33:36   41358      MOP
336     33270      1529 2019-03-23 21:44:00   32594      CUC

      customer_present expiration_date          provider       lat        lng  \
id
872               True           08/27  American Express   5.47090  100.24529
1477              <NA>             NaN               NaN       NaN        NaN
158               True           09/20  American Express  26.26490   81.54855
808               True           04/29     VISA 13 digit  59.37722   28.19028
336              False           04/22        Mastercard  51.39323    0.47713

             region country  no_variance  mostly_nulls
id
872   Batu Feringgi      MY            1          <NA>
1477            NaN     NaN         <NA>          <NA>
158            Jais      IN            1          <NA>
808           Narva      EE            1          <NA>
336          Strood      GB            1          <NA>
```

We can now run search on `X_train_no_errors` and `y_train_no_errors`. Note that the search here doesn't fail since we addressed the errors, but there will still exist warnings in the returned tuple. This search allows the `mostly_nulls` column to remain in the features during search.

[13]:
```
results_no_errors = search_iterative(
    X_train_no_errors, y_train_no_errors, problem_type="binary"
)
results_no_errors
```

[13]:
```
(<evalml.automl.automl_search.AutoMLSearch at 0x7f9c568843d0>,
 [{'message': '1 out of 1200 rows are 95.0% or more null',
   'data_check_name': 'NullDataCheck',
   'level': 'warning',
   'details': {'columns': None,
    'rows': [1477],
    'pct_null_cols': id
    1477    1.0
    dtype: float64},
   'code': 'HIGHLY_NULL_ROWS',
   'action_options': [{'code': 'DROP_ROWS',
     'data_check_name': 'NullDataCheck',
     'metadata': {'columns': None, 'rows': [1477]},
     'parameters': {}}]},
  {'message': "Column(s) 'mostly_nulls' are 95.0% or more null",
   'data_check_name': 'NullDataCheck',
   'level': 'warning',
   'details': {'columns': ['mostly_nulls'],
    'rows': None,
    'pct_null_rows': {'mostly_nulls': 0.9966666666666667}},
   'code': 'HIGHLY_NULL_COLS',
   'action_options': [{'code': 'DROP_COL',
     'data_check_name': 'NullDataCheck',
     'metadata': {'columns': ['mostly_nulls'], 'rows': None},
     'parameters': {}}]},
  {'message': "'no_variance' has 1 unique value.",
   'data_check_name': 'NoVarianceDataCheck',
   'level': 'warning',
   'details': {'columns': ['no_variance'], 'rows': None},
   'code': 'NO_VARIANCE',
   'action_options': [{'code': 'DROP_COL',
     'data_check_name': 'NoVarianceDataCheck',
     'metadata': {'columns': ['no_variance'], 'rows': None},
     'parameters': {}}]}])
```

## 4.9 Utilities

### 4.9.1 Configuring Logging

EvalML uses the standard Python logging package. Default logging behavior prints WARNING level logs and above (ERROR and CRITICAL) to stdout. To configure different behavior, please refer to the Python logging documentation.

To see up-to-date feedback as `AutoMLSearch` runs, use the argument `verbose=True` when instantiating the object. This will temporarily set up a logging object to print INFO level logs and above to stdout, as well as display a graph of the best score over pipeline iterations.

### 4.9.2 System Information

EvalML provides a command-line interface (CLI) tool prints the version of EvalML and core dependencies installed, as well as some basic system information. To use this tool, just run `evalml info` in your shell or terminal. This could be useful for debugging purposes or tracking down any version-related issues.

[1]:
```
!evalml info
```

```
/bin/sh: 1: evalml: not found
```

## 4.10 AutoMLSearch for time series problems

In this guide, we'll show how you can use EvalML to perform an automated search of machine learning pipelines for time series problems. Time series support is still being actively developed in EvalML so expect this page to improve over time.

### 4.10.1 But first, what is a time series?

A time series is a series of measurements taken at different moments in time (Wikipedia). The main difference between a time series dataset and a normal dataset is that the rows of a time series dataset are ordered chronologically, where the relative time between rows is significant. This relationship between the rows does not exist in non-time series datasets. In a non-time-series dataset, you can shuffle the rows and the dataset still has the same meaning. If you shuffle the rows of a time series dataset, the relationship between the rows is completely different!

### 4.10.2 What does AutoMLSearch for time series do?

In a machine learning setting, we are usually interested in using past values of the time series to predict future values. That is what EvalML's time series functionality is built to do.

### 4.10.3 Loading the data

In this guide, we work with daily minimum temperature recordings from Melbourne, Austrailia from the beginning of 1981 to end of 1990.

We start by loading the temperature data into two splits. The first split will be a training split consisting of data from 1981 to end of 1989. This is the data we'll use to find the best pipeline with AutoML. The second split will be a testing split consisting of data from 1990. This is the split we'll use to evaluate how well our pipeline generalizes on unseen data.

[2]:
```
import pandas as pd
from evalml.demos import load_weather

X, y = load_weather()
```

```
             Number of Features
Categorical                   1

Number of training examples: 3650
Targets
10.0    1.40%
```

(continues on next page)

(continued from previous page)

```
11.0    1.40%
13.0    1.32%
12.5    1.21%
10.5    1.21%
        ...
0.2     0.03%
24.0    0.03%
25.2    0.03%
22.7    0.03%
21.6    0.03%
Name: Temp, Length: 229, dtype: object
```

[3]:
```
train_dates, test_dates = X.Date < "1990-01-01", X.Date >= "1990-01-01"
X_train, y_train = X.ww.loc[train_dates], y.ww.loc[train_dates]
X_test, y_test = X.ww.loc[test_dates], y.ww.loc[test_dates]
```

### Visualizing the training set

[4]:
```
import plotly.graph_objects as go
```

[5]:
```
data = [
    go.Scatter(
        x=X_train["Date"],
        y=y_train,
        mode="lines+markers",
        name="Temperature (C)",
        line=dict(color="#1f77b4"),
    )
]
# Let plotly pick the best date format.
layout = go.Layout(
    title={"text": "Min Daily Temperature, Melbourne 1980-1989"},
    xaxis={"title": "Time"},
    yaxis={"title": "Temperature (C)"},
)

go.Figure(data=data, layout=layout)
```

Data type cannot be displayed: application/vnd.plotly.v1+json, text/html

### 4.10.4 Fixing the data

Sometimes, the datasets we work with do not have perfectly consistent DateTime columns. We can use the `TimeSeriesRegularizer` and `TimeSeriesImputer` to correct any discrepancies in our data in a time-series specific way.

To show an example of this, let's create some discrepancies in our training data. We'll also add a couple of extra columns in the X DataFrame to highlight more of the options with these tools.

[6]:
```
X["Categorical"] = [str(i % 4) for i in range(len(X))]
X["Categorical"] = X["Categorical"].astype("category")
X["Numeric"] = [i for i in range(len(X))]

# Re-split the data since we modified X
X_train, y_train = X.loc[train_dates], y.ww.loc[train_dates]
X_test, y_test = X.loc[test_dates], y.ww.loc[test_dates]
```

[7]:
```
X_train["Date"][500] = None
X_train["Date"][1042] = None
X_train["Date"][1043] = None
X_train["Date"][231] = pd.Timestamp("1981-08-19")

X_train.drop(1209, inplace=True)
X_train.drop(398, inplace=True)
y_train.drop(1209, inplace=True)
y_train.drop(398, inplace=True)
```

With these changes, there are now NaN values in the training data that our models won't be able to recognize, and there is no longer a clear frequency between the dates.

[8]:
```
print(f"Inferred frequency: {pd.infer_freq(X_train['Date'])}")
print(f"NaNs in date column: {X_train['Date'].isna().any()}")
print(
    f"NaNs in other training data columns: {X_train['Categorical'].isna().any() or X_
↪train['Numeric'].isna().any()}"
)
print(f"NaNs in target data: {y_train.isna().any()}")
```

```
Inferred frequency: None
NaNs in date column: True
NaNs in other training data columns: False
NaNs in target data: False
```

#### Time Series Regularizer

We can use the `TimeSeriesRegularizer` component to restore the missing and NaN DateTime values we've created in our data. This component is designed to infer the proper frequency using Woodwork's "infer_frequency" function and generate a new DataFrame that follows it. In order to maintain as much original information from the input data as possible, all rows with completely correct times are ported over into this new DataFrame. If there are any rows that have the same timestamp as another, these will be dropped. The first occurrence of a date or time maintains priority. If there are any values that don't quite line up with the inferred frequency they will be shifted to any closely missing datetimes, or dropped if there are none nearby.

[9]:
```
from evalml.pipelines.components import TimeSeriesRegularizer

regularizer = TimeSeriesRegularizer(time_index="Date")
X_train, y_train = regularizer.fit_transform(X_train, y_train)
```

Now we can see that pandas has successfully inferred the frequency of the training data, and there are no more null values within `X_train`. However, by adding values that were dropped before, we have created NaN values within the target data, as well as the other columns in our training data.

[10]:
```
print(f"Inferred frequency: {pd.infer_freq(X_train['Date'])}")
print(f"NaNs in training data: {X_train['Date'].isna().any()}")
print(
    f"NaNs in other training data columns: {X_train['Categorical'].isna().any() or X_
↪train['Numeric'].isna().any()}"
)
print(f"NaNs in target data: {y_train.isna().any()}")
```

```
Inferred frequency: D
NaNs in training data: False
NaNs in other training data columns: True
NaNs in target data: True
```

### Time Series Imputer

We could easily use the `Imputer` and `TargetImputer` components to fill in the missing gaps in our `X` and `y` data. However, these tools are not built for time series problems. Their supported imputation strategies of "mean", "most_frequent", or similar are all static. They don't account for the passing of time, and how neighboring data points may have more predictive power than simply taking the average. The `TimeSeriesImputer` solves this problem by offering three different imputation strategies: - "forwards_fill": fills in any NaN values with the same value as found in the previous non-NaN cell. - "backwards_fill": fills in any NaN values with the same value as found in the next non-NaN cell. - "interpolate": (numeric columns only) fills in any NaN values by linearly interpolating the values of the previous and next non-NaN cells.

[11]:
```
from evalml.pipelines.components import TimeSeriesImputer

ts_imputer = TimeSeriesImputer(
    categorical_impute_strategy="forwards_fill",
    numeric_impute_strategy="backwards_fill",
    target_impute_strategy="interpolate",
)
X_train, y_train = ts_imputer.fit_transform(X_train, y_train)
```

Now, finally, we have a DataFrame that's back in order without flaws, which we can use for running AutoMLSearch and running models without issue.

[12]:
```
print(f"Inferred frequency: {pd.infer_freq(X_train['Date'])}")
print(f"NaNs in training data: {X_train['Date'].isna().any()}")
print(
    f"NaNs in other training data columns: {X_train['Categorical'].isna().any() or X_
↪train['Numeric'].isna().any()}"
)
print(f"NaNs in target data: {y_train.isna().any()}")
```

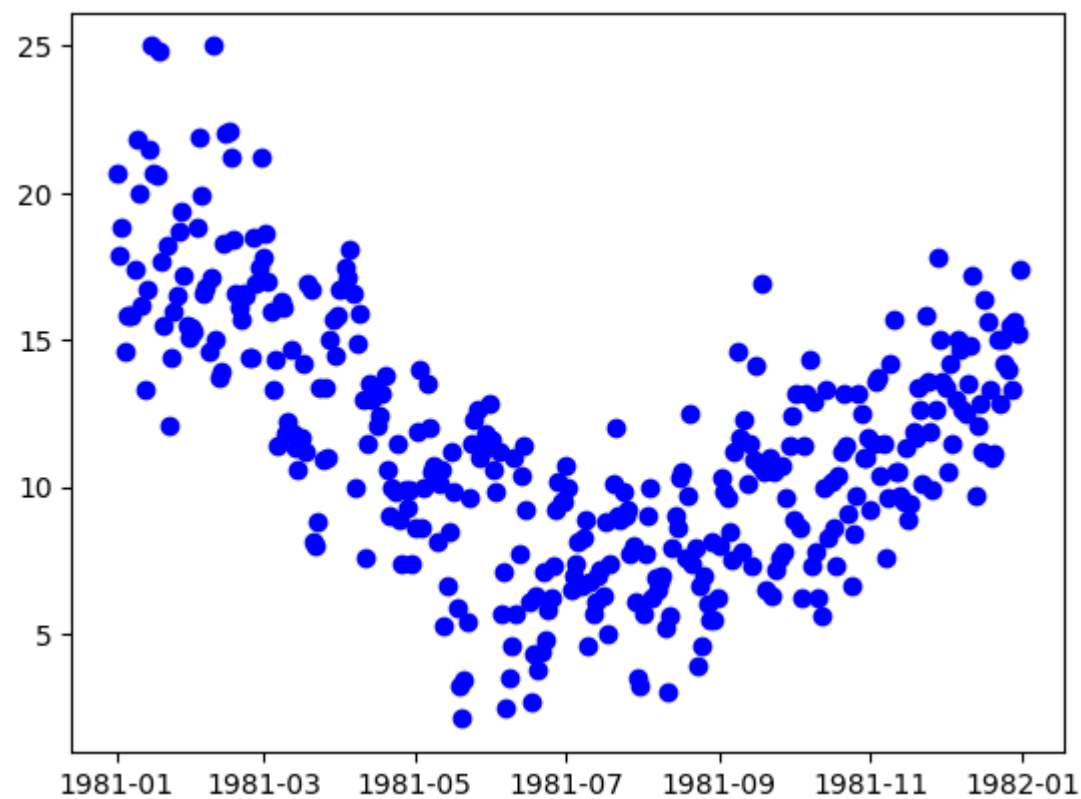
```
Inferred frequency: D
NaNs in training data: False
NaNs in other training data columns: False
NaNs in target data: False
```

### 4.10.5 Trending and Seasonality Decomposition

Decomposing a target signal into a trend and/or a cyclical signal is a common pre-processing step for time series modeling. Having an understanding of the presence or absence of these component signals can provide additional insight and decomposing the signal into these constituent components can enable non-time-series aware estimators to perform better while attempting to model this data. We have two unique decomposers, the `PolynomialDecompser` and the `STLDecomposer`.

Let's first take a look at a year's worth of the weather dataset.

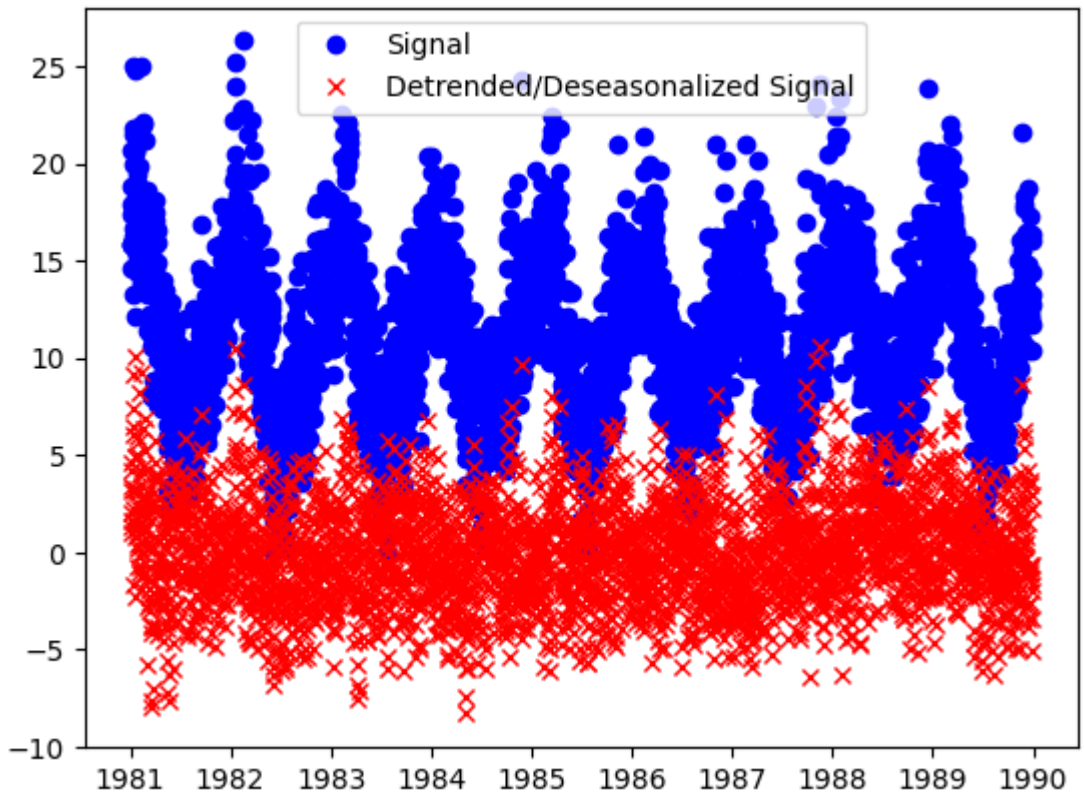
[13]:
```
import matplotlib.pyplot as plt

length = 365
X_train_time = X_train.set_index("Date").asfreq(pd.infer_freq(X_train["Date"]))
y_train_time = y_train.set_axis(X_train["Date"]).asfreq(pd.infer_freq(X_train["Date"]))
plt.plot(y_train_time[0:length], "bo")
plt.show()
```




With the knowledge that this is a weather dataset and the data itself is daily weather data, we can assume that the seasonal data will have a period of approximately 365 data points. Let's build and fit decomposers to detrend and deseasonalize this data.

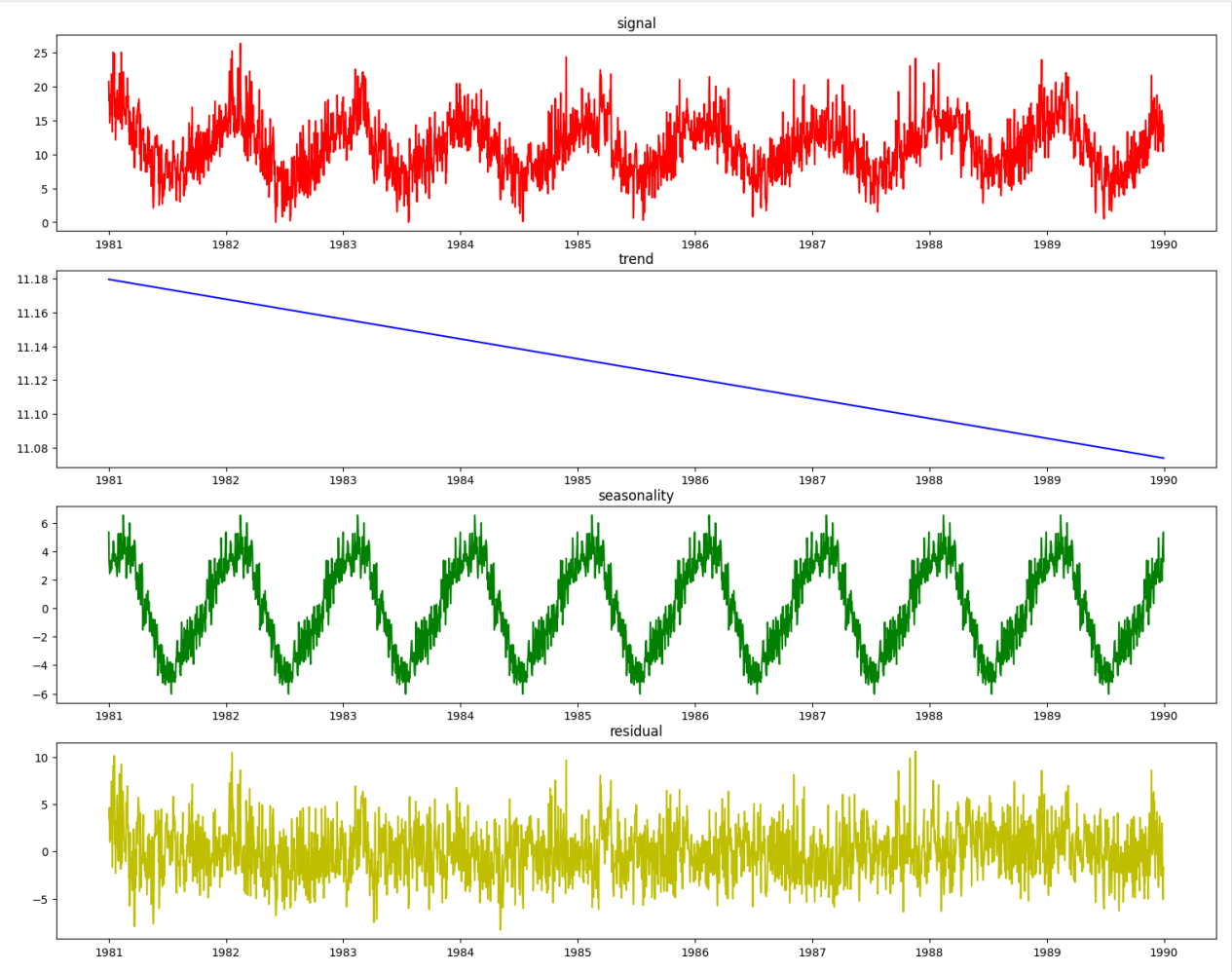
### Polynomial Decomposer

[14]:
```
from evalml.pipelines.components.transformers.preprocessing.polynomial_decomposer import␣
↪(
    PolynomialDecomposer,
)

pdc = PolynomialDecomposer(degree=1, period=365)
X_t, y_t = pdc.fit_transform(X_train_time, y_train_time)

plt.plot(y_train_time, "bo", label="Signal")
plt.plot(y_t, "rx", label="Detrended/Deseasonalized Signal")
plt.legend()
plt.show()
```




The result is the residual signal, with the trend and seasonality removed. If we want to look at what the component identified as the trend and seasonality, we can call the `plot_decomposition()` function.

[15]:
```
res = pdc.plot_decomposition(X_train_time, y_train_time)
```




It is desirable to enhance the decomposer component with a guess at the period of the seasonal aspect of the signal before decomposing it. To do that, we can use the `determine_periodicity(X, y)` function of the `Decomposer` class.

[16]:
```
period = pdc.determine_periodicity(X_train_time, y_train_time)
print(period)
```

```
351
```

The `PolynomialDecomposer` class, if not explicitly set in the constructor, will set its `period` parameter based on a `statsmodels` function `freq_to_period` that considers the frequency of the datetime data. This will give a reasonable guess as to what the frequency could be. For example, if the `PolynomialDecomposer` object is fit with `period` not explicitly set, it will take on a default value of 7, which is good for daily data signals that have a known seasonal component period that is weekly.

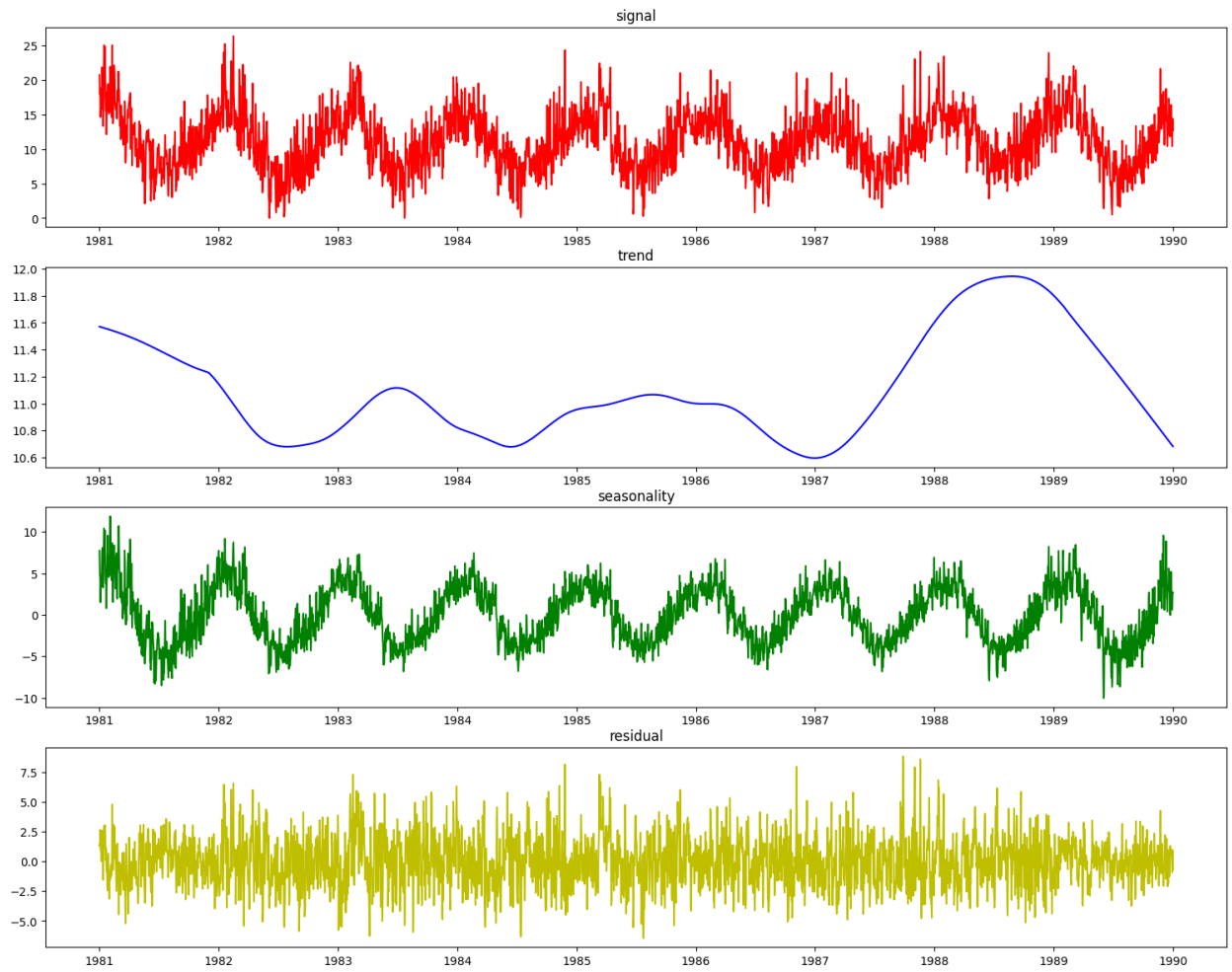
In this case where the seasonal period is not known beforehand, the `set_period()` convenience function will look at the target data, determine a better guess for the period and set the `Decomposer` object appropriately.

[17]:
```
pdc = PolynomialDecomposer()
pdc.fit(X_train_time, y_train_time)
assert pdc.period == 7
pdc.set_period(X_train_time, y_train_time)
assert 350 < pdc.period < 370
```

### STLDecomposer

The `STLDecomposer` runs on statsmodels' implementation of STL decomposition. Let's take a look at how STL decomposes the weather dataset.

[18]:
```
from evalml.pipelines.components import STLDecomposer

stl = STLDecomposer()
X_t, y_t = stl.fit_transform(X_train_time, y_train_time)

res = stl.plot_decomposition(X_train_time, y_train_time)
```




This doesn't look nearly as good as the `PolynomialDecomposer` did. This is because STL decomposition performs best when the data has a small seasonal period, generally less than 14 time units. The weather dataset's seasonal period of ~365 days does not work as well since STL extracted a shorter term seasonality for decomposition.

We can generate some synthetic data that better highlights where STL performs well. For this example, we'll generate monthly data with an annual seasonal period.

[19]:
```
import random
import numpy as np
from datetime import datetime
from sklearn.preprocessing import minmax_scale
```

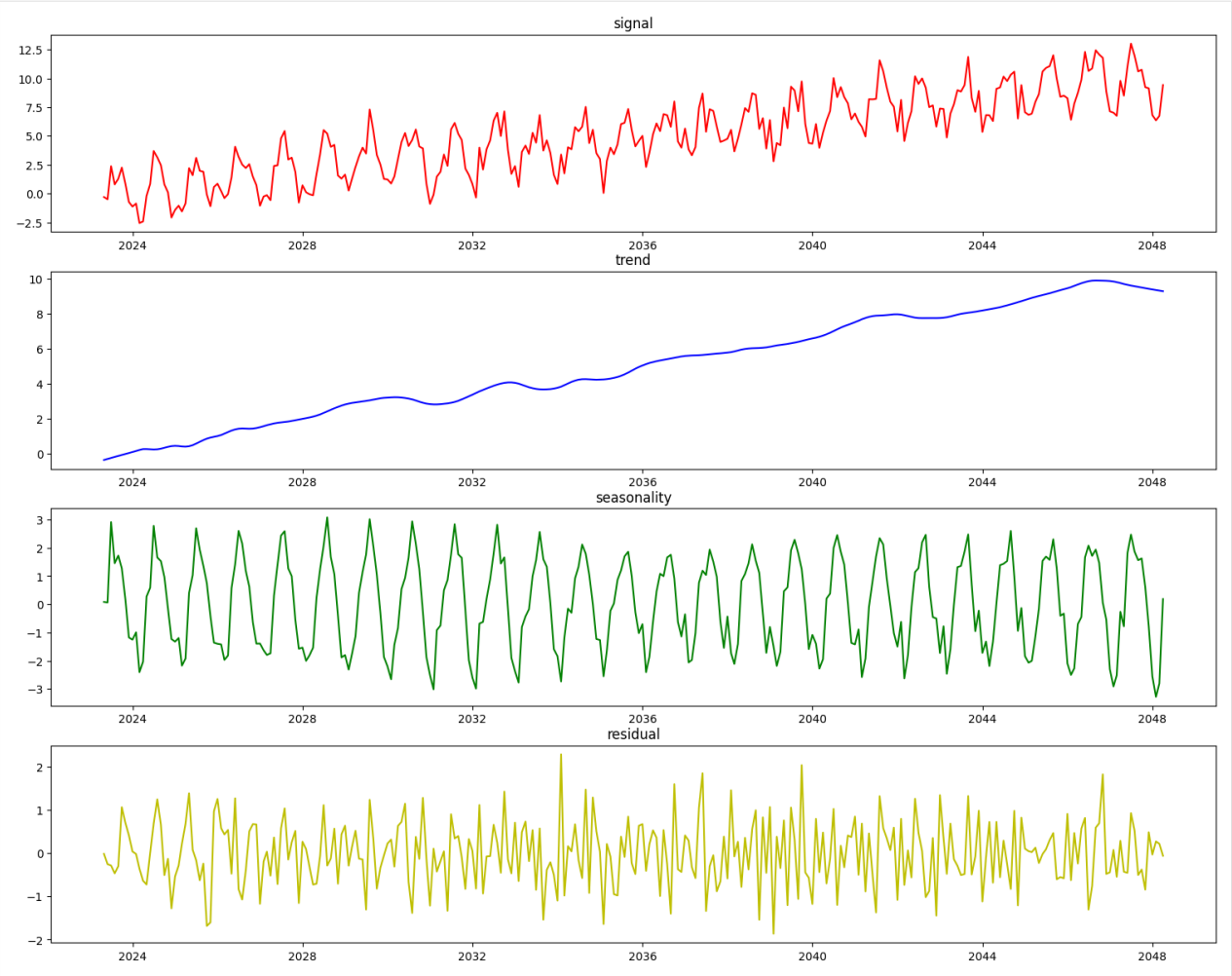
(continues on next page)

(continued from previous page)

```
def generate_synthetic_data(
    period=12,
    num_periods=25,
    scale=10,
    seasonal_scale=2,
    trend_degree=1,
    freq_str="M",
):
    freq = 2 * np.pi / period
    x = np.arange(0, period * num_periods, 1)
    dts = pd.date_range(datetime.today(), periods=len(x), freq=freq_str)
    X = pd.DataFrame({"x": x})
    X = X.set_index(dts)

    if trend_degree == 1:
        y_trend = pd.Series(scale * minmax_scale(x + 2))
    elif trend_degree == 2:
        y_trend = pd.Series(scale * minmax_scale(x**2))
    elif trend_degree == 3:
        y_trend = pd.Series(scale * minmax_scale((x - 5) ** 3 + x**2))
    y_seasonal = pd.Series(seasonal_scale * np.sin(freq * x))
    y_random = pd.Series(np.random.normal(0, 1, len(X)))
    y = y_trend + y_seasonal + y_random
    return X, y


X_stl, y_stl = generate_synthetic_data()
```

Let's see how the `STLDecomposer` does at decomposing this data.

[20]:
```
stl = STLDecomposer()
X_t_stl, y_t_stl = stl.fit_transform(X_stl, y_stl)

res = stl.plot_decomposition(X_stl, y_stl)
```




On top of decomposing this type of data well, the statsmodels implementation of STL automatically determines the seasonal period of the data, which is saved during fit time for this component.

[21]:
```
stl = STLDecomposer()
assert stl.period == None
stl.fit(X_stl, y_stl)
print(stl.period)
```

```
12
```

### 4.10.6 Running AutoMLSearch

`AutoMLSearch` for time series problems works very similarly to the other problem types with the exception that users need to pass in a new parameter called `problem_configuration`.

The `problem_configuration` is a dictionary specifying the following values:

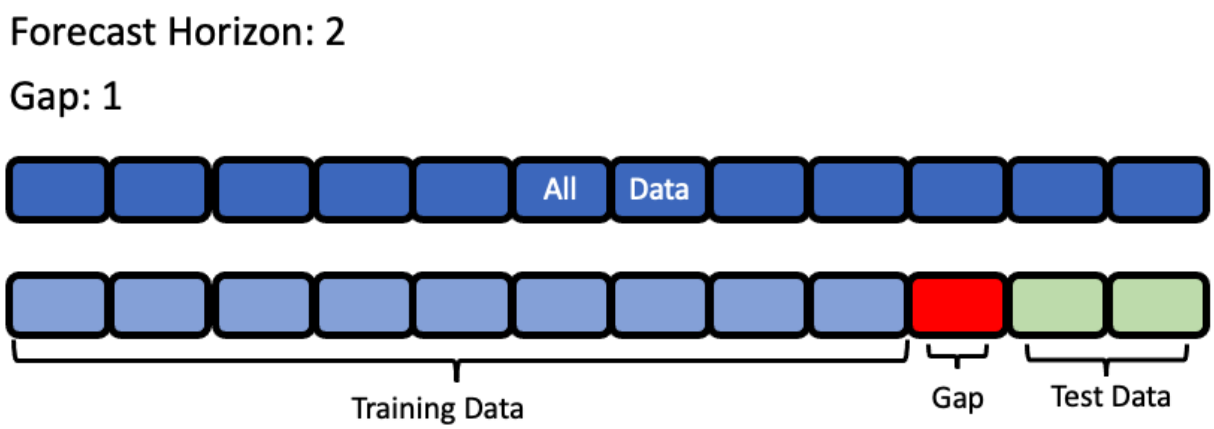
- **forecast_horizon**: The number of time periods we are trying to forecast. In this example, we're interested in predicting weather for the next 7 days, so the value is 7.
- **gap**: The number of time periods between the end of the training set and the start of the test set. For example, in our case we are interested in predicting the weather for the next 7 days with the data as it is "today", so the gap

is 0. However, if we had to predict the weather for next Monday-Sunday with the data as it was on the previous Friday, the gap would be 2 (Saturday and Sunday separate Monday from Friday). It is important to select a value that matches the realistic delay between the forecast date and the most recently avaliable data that can be used to make that forecast.

- **max_delay**: The maximum number of rows to look in the past from the current row in order to compute features. In our example, we'll say we can use the previous week's weather to predict the current week's.
- **time_index**: The column of the training dataset that contains the date corresponding to each observation. While only some of the models we run during time series searches require the `time_index`, we require it to be passed in to top level search so that the parameter can reach the models that need it.

Note that the values of these parameters must be in the same units as the training/testing data.

### Visualization of forecast horizon and gap




[22]:
```
from evalml.automl import AutoMLSearch

problem_config = {"gap": 0, "max_delay": 7, "forecast_horizon": 7, "time_index": "Date"}

automl = AutoMLSearch(
    X_train,
    y_train,
    problem_type="time series regression",
    max_batches=1,
    problem_configuration=problem_config,
    automl_algorithm="iterative",
    allowed_model_families=[
        "xgboost",
        "random_forest",
        "linear_model",
        "extra_trees",
        "decision_tree",
    ],
)
```

[23]:
```
automl.search()
```

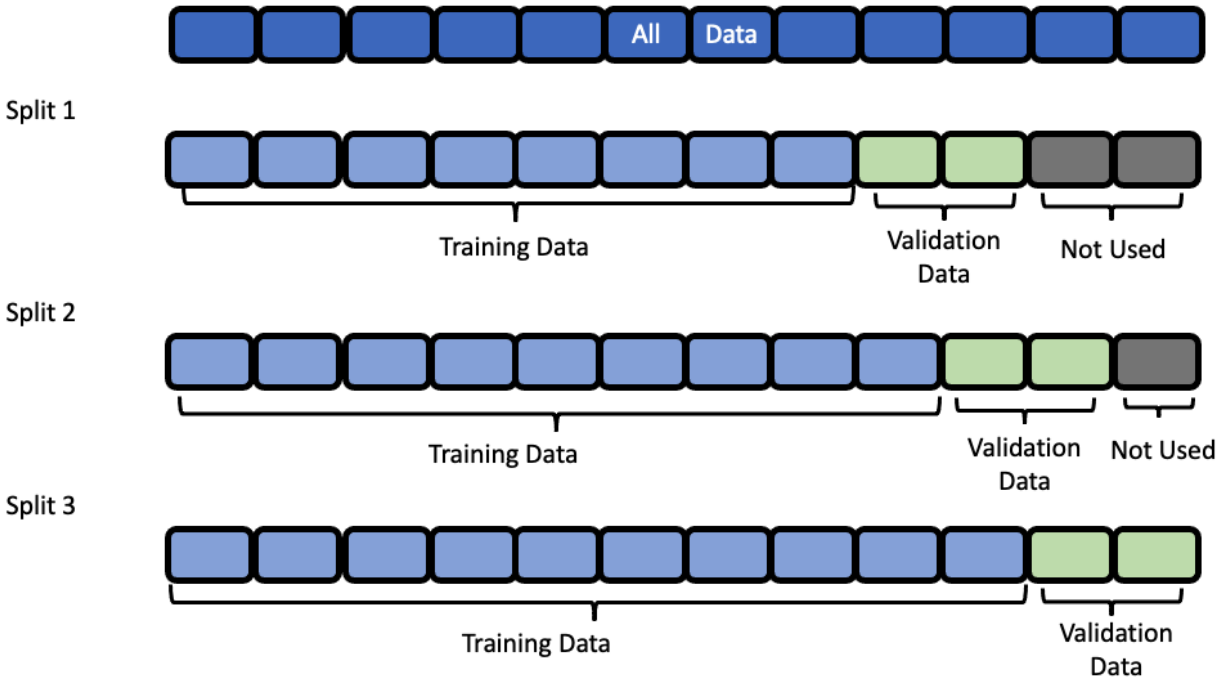
[23]:
```
{1: {'Elastic Net Regressor w/ Imputer + Time Series Featurizer + STL Decomposer +␣
↪DateTime Featurizer + One Hot Encoder + Drop NaN Rows Transformer + Standard Scaler':
↪'00:25',
 'Elastic Net Regressor w/ Imputer + Time Series Featurizer + DateTime Featurizer + One␣
↪Hot Encoder + Drop NaN Rows Transformer + Standard Scaler': '00:03',
 'XGBoost Regressor w/ Imputer + Time Series Featurizer + STL Decomposer + DateTime␣
↪Featurizer + One Hot Encoder': '00:27',
 'XGBoost Regressor w/ Imputer + Time Series Featurizer + DateTime Featurizer + One Hot␣
↪Encoder': '00:05',
 'Random Forest Regressor w/ Imputer + Time Series Featurizer + STL Decomposer +␣
↪DateTime Featurizer + One Hot Encoder + Drop NaN Rows Transformer': '00:27',
 'Random Forest Regressor w/ Imputer + Time Series Featurizer + DateTime Featurizer +␣
↪One Hot Encoder + Drop NaN Rows Transformer': '00:05',
 'Decision Tree Regressor w/ Imputer + Time Series Featurizer + STL Decomposer +␣
↪DateTime Featurizer + One Hot Encoder + Drop NaN Rows Transformer': '00:24',
 'Decision Tree Regressor w/ Imputer + Time Series Featurizer + DateTime Featurizer +␣
↪One Hot Encoder + Drop NaN Rows Transformer': '00:02',
 'Extra Trees Regressor w/ Imputer + Time Series Featurizer + STL Decomposer + DateTime␣
↪Featurizer + One Hot Encoder + Drop NaN Rows Transformer': '00:25',
 'Extra Trees Regressor w/ Imputer + Time Series Featurizer + DateTime Featurizer + One␣
↪Hot Encoder + Drop NaN Rows Transformer': '00:03',
 'Total time of batch': '02:32'}}
```

### 4.10.7 Understanding what happened under the hood

This is great, `AutoMLSearch` is able to find a pipeline that scores an R2 value of 0.44 compared to a baseline pipeline that is only able to score 0.07. But how did it do that?

#### Data Splitting

EvalML uses rolling origin cross validation for time series problems. Basically, we take successive cuts of the training data while keeping the validation set size fixed at `forecast_horizon` number of time units. Note that the splits are not separated by `gap` number of units. This is because we need access to all the data to generate features for every row of the validation set. However, the feature engineering done by our pipelines respects the `gap` value. This is explained more in the *feature engineering section*.




### Baseline Pipeline

The most naive thing we can do in a time series problem is use the most recently available observation to predict the next observation. In our example, this means we'll use the measurement from 7 days ago as the prediction for the current date.

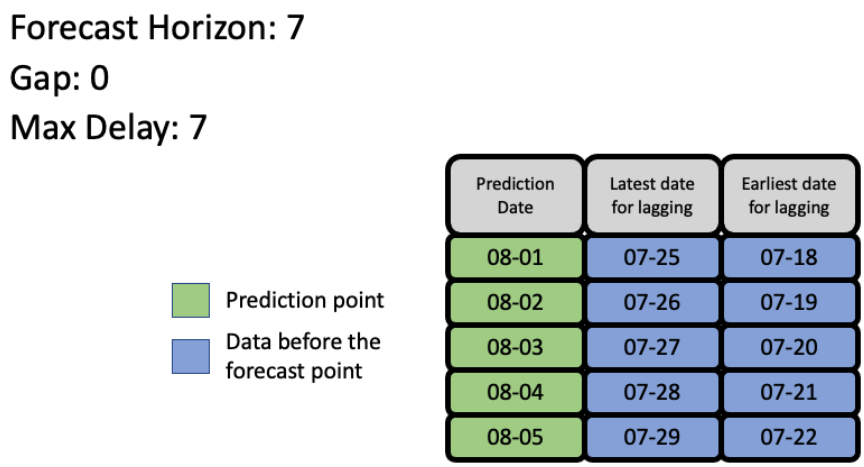
[24]:
```
import pandas as pd

baseline = automl.get_pipeline(0)
baseline.fit(X_train, y_train)
naive_baseline_preds = baseline.predict_in_sample(
    X_test, y_test, objective=None, X_train=X_train, y_train=y_train
)
expected_preds = pd.Series(
    pd.concat([y_train.iloc[-7:], y_test]).shift(7).iloc[7:], name="target"
)
pd.testing.assert_series_equal(expected_preds, naive_baseline_preds)
```

### Feature Engineering

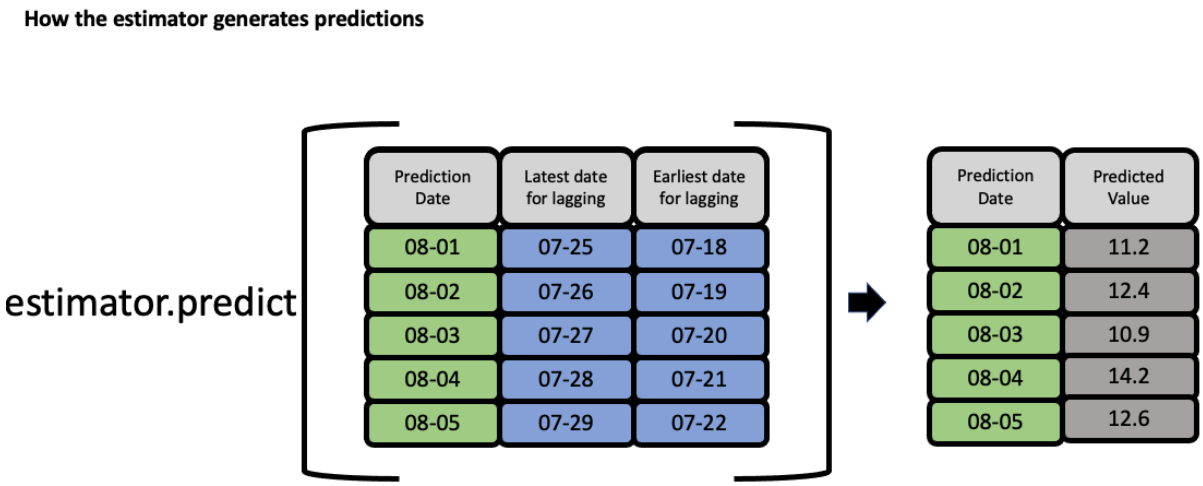
EvalML uses the values of `gap`, `forecast_horizon`, and `max_delay` to calculate a "window" of allowed dates that can be used for engineering the features of each row in the validation/test set. The formula for computing the bounds of the window is:

[t - (max_delay + forecast_horizon + gap), t - (forecast_horizon + gap)]

As an example, this is what the features for the first five days of August would look like in our current problem:




The estimator then takes these features to generate predictions:




### Feature engineering components for time series

For an example of a time-series feature engineering component see *TimeSeriesFeaturizer*

## 4.10.8 Evaluate best pipeline on test data

Now that we have covered the mechanics of how EvalML runs AutoMLSearch for time series pipelines, we can compare the performance on the test set of the best pipeline found during search and the baseline pipeline.

[25]:
```
pl = automl.best_pipeline

pl.fit(X_train, y_train)

best_pipeline_score = pl.score(X_test, y_test, ["MedianAE"], X_train, y_train)[
    "MedianAE"
]
```

[26]:
```
best_pipeline_score
```

[26]:
```
1.903458595275879
```

[27]:
```
baseline = automl.get_pipeline(0)
baseline.fit(X_train, y_train)
naive_baseline_score = baseline.score(X_test, y_test, ["MedianAE"], X_train, y_train)[
    "MedianAE"
]
```

[28]:
```
naive_baseline_score
```

[28]:
```
2.3
```

The pipeline found by AutoMLSearch has a 31% improvement over the naive forecast!

[29]:
```
automl.objective.calculate_percent_difference(best_pipeline_score, naive_baseline_score)
```

[29]:
```
17.240930640179172
```

### 4.10.9 Visualize the predictions over time

[30]:
```
from evalml.model_understanding import graph_prediction_vs_actual_over_time

fig = graph_prediction_vs_actual_over_time(
    pl, X_test, y_test, X_train, y_train, dates=X_test["Date"]
)
fig
```

Data type cannot be displayed: application/vnd.plotly.v1+json, text/html

### 4.10.10 Predicting on unseen data

You'll notice that in the code snippets here, we use the `predict_in_sample` pipeline method as opposed to the usual `predict` method. What's the difference?

- `predict_in_sample` is used when the target value is known on the dates we are predicting on. This is true in cross validation. This method has an expected `y` parameter so that we can compute features using previous target values for all of the observations on the holdout set.
- `predict` is used when the target value is not known, e.g. the test dataset. The y parameter is not expected as only the target is observed in the training set. The test dataset must be separated by `gap` units from the training dataset. For the moment, the test set size must be less than or equal to `forecast_horizon`.

Here is an example of these two methods in action:

#### predict_in_sample

[31]:
```
pl.predict_in_sample(X_test, y_test, objective=None, X_train=X_train, y_train=y_train)
```

[31]:
```
3287    12.007137
3288    12.502100
3289    12.578979
3290    11.418142
3291    11.636833
          ...
3647    13.354449
3648    13.750842
3649    13.747188
3650    14.131168
3651    12.356060
Name: target, Length: 365, dtype: float64
```

#### predict

[32]:
```
pl.predict(X_test, objective=None, X_train=X_train, y_train=y_train)
```

[32]:
```
3287    12.007137
3288    12.502100
3289    12.578979
3290    11.418142
3291    11.636833
          ...
3647    13.228288
3648    13.290761
3649    13.062471
3650    13.233994
3651    14.117554
Name: target, Length: 365, dtype: float64
```

### 4.10.11 Validating the holdout data

Before we predict on our holdout data, it is important to validate that it meets the requirements we summarized in the second point above in **Predicting on unseen data**. We can call on `validate_holdout_datasets` in order to verify the two requirements:

1. The holdout data is separated by `gap` units from the training dataset. This is determined by the `time_index` column, not the index e.g. if your datetime frequency for the column "Date" is 2 days with a `gap` of 3, then the holdout data must start 2 days x 3 = 6 days after the training data.
2. The length of the holdout data must be less than or equal to the `forecast_horizon`.

[33]:
```
from evalml.utils.gen_utils import validate_holdout_datasets

# Holdout dataset has 365 observations
validation_results = validate_holdout_datasets(X_test, X_train, problem_config)
assert not validation_results.is_valid
# Holdout dataset has 7 observations
```

(continues on next page)

(continued from previous page)

```
validation_results = validate_holdout_datasets(
    X_test.iloc[: pl.forecast_horizon], X_train, problem_config
)
assert validation_results.is_valid
```

### predict – Test set size matches forecast horizon

[34]:
```
pl.predict(
    X_test.iloc[: pl.forecast_horizon], objective=None, X_train=X_train, y_train=y_train
)
```

[34]:
```
3287    12.007137
3288    12.502100
3289    12.578979
3290    11.418142
3291    11.636833
3292    11.532094
3293    12.126741
Name: target, dtype: float64
```

### predict – Test set size is less than forecast horizon

[35]:
```
pl.predict(
    X_test.iloc[: pl.forecast_horizon - 2],
    objective=None,
    X_train=X_train,
    y_train=y_train,
)
```

[35]:
```
3287    12.007137
3288    12.502100
3289    12.578979
3290    11.418142
3291    11.636833
Name: target, dtype: float64
```

### predict – Test set size index starts at 0

[36]:
```
pl.predict(
    X_test.iloc[: pl.forecast_horizon].reset_index(drop=True),
    objective=None,
    X_train=X_train,
    y_train=y_train,
)
```

[36]:
```
3287    12.007137
3288    12.502100
3289    12.578979
3290    11.418142
```

(continues on next page)

(continued from previous page)

```
3291    11.636833
3292    11.532094
3293    12.126741
Name: target, dtype: float64
```

### 4.10.12 Prediction Intervals

#### Getting Prediction Intervals

While predictions that are generated by EvalML pipelines aim to be accurate as possible, it is very rarely the case that future results are the exact same values as predicted. Prediction intervals can help to contextualize a prediction by showing the range a future prediction is expected to fall within a certain likelihood.

Given the preprocessed (transformed, ready for prediction) features, the corresponding predictions, and a **fitted** EvalML estimator, the prediction intervals for this set of predictions is generated by calling `get_prediction_intervals()` on the pipeline's estimator. Here, we use the fitted estimator in our trained EvalML pipeline to generate the prediction intervals:

[37]:
```
X_trans = pl.transform_all_but_final(X_test, y_test)
y_pred = pl.predict(X_test, objective=None, X_train=X_train, y_train=y_train)
pl.estimator.get_prediction_intervals(X=X_trans, y=y_pred)
```

[37]:
```
{'0.95_lower': 3287    16.468615
 3288    16.504353
 3289    16.688278
 3290    16.811346
 3291    16.942591
           ...
 3647    12.070903
 3648    12.418325
 3649    12.379229
 3650    12.787175
 3651    11.024206
 Length: 365, dtype: float64,
 '0.95_upper': 3287    17.256910
 3288    17.292648
 3289    17.476574
 3290    17.599642
 3291    17.730886
           ...
 3647    14.637996
 3648    15.083359
 3649    15.115146
 3650    15.475162
 3651    13.687914
 Length: 365, dtype: float64}
```

By default, prediction intervals are calculated for the 95% upper and lower bound. In the above example, 95% of the time, a prediction sometime in the future will fall in this range.

To generate prediction intervals for a custom value, use the `coverage` parameter. In the example below, the 80% interval range is calculated below:

[38]:
```
pl.estimator.get_prediction_intervals(X=X_trans, y=y_pred, coverage=[0.8])
```

[38]:
```
{'0.8_lower': 3287    16.605043
 3288    16.640781
 3289    16.824707
 3290    16.947775
 3291    17.079019
           ...
 3647    12.515183
 3648    12.879556
 3649    12.852728
 3650    13.252378
 3651    11.485208
 Length: 365, dtype: float64,
 '0.8_upper': 3287    17.120482
 3288    17.156220
 3289    17.340145
 3290    17.463213
 3291    17.594458
           ...
 3647    14.193715
 3648    14.622128
 3649    14.641648
 3650    15.009959
 3651    13.226912
 Length: 365, dtype: float64}
```

### 4.10.13 Forecasting Future Data

Unlike standard pipelines, time series pipelines are able to generate predictions out to the future. The number of predictions out in the future is dependent on the `forecast_horizon` parameter set in the problem configuration of an AutoML search.

To show that it is possible to generate brand new predictions in the future, the entire weather dataset (including the holdout set) will be used. The code block below refit the pipeline on the entire dataset and generates a forecast.

[39]:
```
X.ww.init()
y.ww.init()

pl.fit(X, y)
X_forecast_dates = pl.get_forecast_period(X=X).to_frame()
y_forecast = pl.get_forecast_predictions(X=X, y=y)
display("Forecast Dates:", X_forecast_dates)
display("Forecast Predictions:", y_forecast)
```

```
'Forecast Dates:'
```

```
           Date
3652 1991-01-01
3653 1991-01-02
3654 1991-01-03
3655 1991-01-04
3656 1991-01-05
```

(continues on next page)

(continued from previous page)

```
3657 1991-01-06
3658 1991-01-07
```

```
'Forecast Predictions:'
```

```
3652    12.260047
3653    10.095071
3654    11.425120
3655    12.398380
3656    12.176962
3657    12.155176
3658    12.144207
Name: Temp, dtype: float64
```

Using these forecasted values, it is possible to generate the prediction intervals for each forecasted point.

[40]:
```
res = pl.get_prediction_intervals(
    X=pd.DataFrame(X_forecast_dates), y=y_forecast, X_train=X, y_train=y
)
display(res)
```

```
{'0.95_lower': 3652    8.390696
 3653    4.622983
 3654    4.723208
 3655    4.659679
 3656    3.524831
 3657    2.677241
 3658    1.906867
 Name: 0.95_lower, dtype: float64,
 '0.95_upper': 3652    16.129398
 3653    15.567159
 3654    18.127032
 3655    20.137082
 3656    20.829093
 3657    21.633111
 3658    22.381547
 Name: 0.95_upper, dtype: float64}
```

[41]:
```
y_lower = res["0.95_lower"]
y_upper = res["0.95_upper"]
```

Using the forecasted predictions and corresponding prediction intervals, we can plot this data. For this plot, only the last 31 days of data will be used so that the forecasted data is visible.

[42]:
```
X_before = X[-31:]
y_before = y[-31:]
```

[43]:
```
fig = go.Figure(
    [
        # Plot last 31 days of training data
        go.Scatter(x=X_before["Date"], y=y_before, name="Training Data", mode="lines"),
        # Plot forecast data
        go.Scatter(
```

(continues on next page)

(continued from previous page)

```
            x=X_forecast_dates["Date"], y=y_forecast, name="Forecast Data", mode="lines"
        ),
        # Plot prediction intervals
        go.Scatter(
            x=X_forecast_dates["Date"].append(X_forecast_dates["Date"][::-1]),
            y=y_upper.append(y_lower[::-1]),
            fill="toself",
            fillcolor="rgba(255,0,0,0.2)",
            line=dict(color="rgba(255,0,0,0.2)"),
            name="Forecast Prediction Intervals",
            showlegend=True,
        ),
    ],
    layout={
        "title": "Plot of Last Two Weeks of Data + Forecast Data With Prediction␣
↪Intervals",
        "xaxis": dict(title="Date"),
        "yaxis": dict(title="Temperature (C)"),
    },
)
fig.show()
```

Data type cannot be displayed: application/vnd.plotly.v1+json, text/html

### 4.10.14 Forecasting into the future

Our previous examples have shown using a pipeline to predict on data we had at training time. However, we can also use EvalML time series pipelines to forecast dates into the future as long we provide data that meets the requirements of **Predicting on unseen data** as well.

To help figure out the dates we need in `X_train` to forecast dates into the future - we've provided `dates_needed_for_prediction` and `dates_needed_for_prediction_range`.

[44]:
```
forecast_date = pd.Timestamp("1991-01-07")
beginning_date, end_date = pl.dates_needed_for_prediction(forecast_date)

print("Dates needed:")
print(f"{beginning_date.strftime('%Y-%m-%d %X')} to {end_date.strftime('%Y-%m-%d %X')}")
```

```
Dates needed:
1990-12-23 00:00:00 to 1991-01-06 00:00:00
```

We can see how the dates are valid by generating some future dates and features with the above date range.

[45]:
```
import random

dates = pd.date_range(
    beginning_date,
    end_date,
    freq=pl.frequency.split("-")[0],
```

(continues on next page)

(continued from previous page)

```
)

X_train_forecast = pd.DataFrame(index=[i + 1 for i in range(len(dates))])
categorical_feature = pd.Series(
    [random.randint(0, 3) for i in range(len(dates))], index=X_train_forecast.index
)
numeric_feature = pd.Series(
    [i + 1 for i in range(len(dates))], index=X_train_forecast.index
)

X_train_forecast["Date"] = pd.Series(dates.values, index=X_train_forecast.index)
X_train_forecast["Categorical"] = pd.Series(
    categorical_feature.values, index=X_train_forecast.index
)
X_train_forecast["Numeric"] = pd.Series(
    numeric_feature.values, index=X_train_forecast.index
)
X_train_forecast.ww.init(
    logical_types={"Categorical": "categorical", "Numeric": "integer"}
)

y_train_forecast = pd.Series(
    X_train_forecast["Numeric"].values, index=X_train_forecast.index
)
```

[46]:
```
X_test_forecast = pd.DataFrame(
    {"Date": [forecast_date], "Categorical": [3], "Numeric": [53862]}
)
```

… and we succesfully have our prediction!

[47]:
```
pl.predict(X_test_forecast, X_train=X_train_forecast, y_train=y_train_forecast)
```

[47]:
```
16    10.221612
Name: Temp, dtype: float64
```

[48]:
```
forecast_start = pd.Timestamp("1991-01-07")
forecast_end = pd.Timestamp("1991-01-14")

dates = pl.dates_needed_for_prediction_range(forecast_start, forecast_end)
print("Dates needed:")
print(f"{dates[0].strftime('%Y-%m-%d %X')} to {dates[1].strftime('%Y-%m-%d %X')}")
```

```
Dates needed:
1990-12-23 00:00:00 to 1991-01-13 00:00:00
```

### 4.10.15 Known-in-advance features

In time series problems, the goal is to predict an unknown value of a data series corresponding to a future moment in time. Since the state of the world is not known in the future, we create features from data in the past since those values are known when we go make our prediction.

However, there are some features corresponding to dates in the future that can be known with certainty, either because they can be derived from the forecast date or because the feature can be controlled by the modeler. This includes features such as if the date is a US Holiday, or the location of a store in a sales dataset. With these sorts of features, we don't need to include them in our time-series specific preprocessing steps (such as Time Series Featurization).

To handle these features, EvalML will split them into a separate path through the component graph, bypassing the unnecessary preprocessing steps. Let's take a look at what that looks like, using some synthetic data.

[50]:
```
X = pd.DataFrame(
    {"features": range(101, 601), "date": pd.date_range("2010-10-01", periods=500)}
)
y = pd.Series(range(500))

X.ww.init()
X.ww["bool_feature"] = (
    pd.Series([True, False]).sample(n=X.shape[0], replace=True).reset_index(drop=True)
)
X.ww["cat_feature"] = (
    pd.Series(["a", "b", "c"]).sample(n=X.shape[0], replace=True).reset_index(drop=True)
)

automl = AutoMLSearch(
    X,
    y,
    problem_type="time series regression",
    problem_configuration={
        "max_delay": 5,
        "gap": 3,
        "forecast_horizon": 2,
        "time_index": "date",
        "known_in_advance": ["bool_feature", "cat_feature"],
    },
)
automl.search()
```

```
17:02:28 - cmdstanpy - INFO - Chain [1] start processing
17:02:28 - cmdstanpy - INFO - Chain [1] done processing
17:02:29 - cmdstanpy - INFO - Chain [1] start processing
17:02:29 - cmdstanpy - INFO - Chain [1] done processing
17:02:29 - cmdstanpy - INFO - Chain [1] start processing
17:02:29 - cmdstanpy - INFO - Chain [1] done processing
```

[50]:
```
{1: {'Random Forest Regressor w/ Select Columns Transformer + Imputer + Time Series␣
↪Featurizer + DateTime Featurizer + Select Columns Transformer + Imputer + One Hot␣
↪Encoder + Drop NaN Rows Transformer': '00:01',
  'Total time of batch': '00:01'},
 2: {'ARIMA Regressor w/ Select Columns Transformer + Imputer + Time Series Featurizer +␣
↪Select Columns Transformer + Imputer + One Hot Encoder': '00:23',
  'Exponential Smoothing Regressor w/ Select Columns Transformer + Imputer + Time Series␣
↪Featurizer + DateTime Featurizer + Select Columns Transformer + Imputer + One Hot␣
↪Encoder': '00:00',
```

(continues on next page)

(continued from previous page)

```
 'Prophet Regressor w/ Select Columns Transformer + Imputer + Time Series Featurizer +␣
↪Select Columns Transformer + Imputer + One Hot Encoder': '00:01',
 'Decision Tree Regressor w/ Select Columns Transformer + Imputer + Time Series␣
↪Featurizer + DateTime Featurizer + Select Columns Transformer + Imputer + One Hot␣
↪Encoder + Drop NaN Rows Transformer': '00:01',
 'Extra Trees Regressor w/ Select Columns Transformer + Imputer + Time Series␣
↪Featurizer + DateTime Featurizer + Select Columns Transformer + Imputer + One Hot␣
↪Encoder + Drop NaN Rows Transformer': '00:01',
 'XGBoost Regressor w/ Select Columns Transformer + Imputer + Time Series Featurizer +␣
↪DateTime Featurizer + Select Columns Transformer + Imputer + One Hot Encoder': '00:01',
 'CatBoost Regressor w/ Select Columns Transformer + Imputer + Time Series Featurizer +␣
↪DateTime Featurizer + Select Columns Transformer + Imputer': '00:01',
 'LightGBM Regressor w/ Select Columns Transformer + Imputer + Time Series Featurizer +␣
↪DateTime Featurizer + Select Columns Transformer + Imputer + One Hot Encoder': '00:01',
 'Elastic Net Regressor w/ Select Columns Transformer + Imputer + Time Series␣
↪Featurizer + DateTime Featurizer + Standard Scaler + Select Columns Transformer +␣
↪Imputer + One Hot Encoder + Standard Scaler + Drop NaN Rows Transformer': '00:01',
 'Total time of batch': '00:34'}}
```

[51]:
```
pipeline = automl.best_pipeline
pipeline.graph()
```

[51]:

## 4.11 FAQ

### 4.11.1 Q: What is the difference between EvalML and other AutoML libraries?

EvalML optimizes machine learning pipelines on *custom practical objectives* instead of vague machine learning loss functions so that it will find the best pipelines for your specific needs. Furthermore, EvalML *pipelines* are able to take in all kinds of data (missing values, categorical, etc.) as long as the data are in a single table. EvalML also allows you to build your own pipelines with existing or custom components so you can have more control over the AutoML process. Moreover, EvalML also provides you with support in the form of *data checks* to ensure that you are aware of potential issues your data may cause with machine learning algorithms.

### 4.11.2 Q: How does EvalML handle missing values?

EvalML contains imputation components in its pipelines so that missing values are taken care of. EvalML optimizes over different types of imputation to search for the best possible pipeline. You can find more information about components *here* and in the API reference *here*.

### 4.11.3 Q: How does EvalML handle categorical encoding?

EvalML provides a *one-hot-encoding component* in its pipelines for categorical variables. EvalML plans to support other encoders in the future.

### 4.11.4 Q: How does EvalML handle feature selection?

EvalML currently utilizes scikit-learn's SelectFromModel with a Random Forest classifier/regressor to handle feature selection. EvalML plans on supporting more feature selectors in the future. You can find more information in the API reference *here*.

### 4.11.5 Q: How is feature importance calculated?

Feature importance depends on the estimator used. Variable coefficients are used for regression-based estimators (Logistic Regression and Linear Regression) and Gini importance is used for tree-based estimators (Random Forest and XGBoost).

### 4.11.6 Q: How does hyperparameter tuning work?

EvalML tunes hyperparameters for its pipelines through Bayesian optimization. In the future we plan to support more optimization techniques such as random search.

### 4.11.7 Q: Can I create my own objective metric?

Yes you can! You can *create your own custom objective* so that EvalML optimizes the best model for your needs.

### 4.11.8 Q: How does EvalML avoid overfitting?

EvalML provides *data checks* to combat overfitting. Such data checks include detecting label leakage, unstable pipelines, hold-out datasets and cross validation. EvalML defaults to using Stratified K-Fold cross-validation for classification problems and K-Fold cross-validation for regression problems but allows you to utilize your own cross-validation methods as well.

### 4.11.9 Q: Can I create my own pipeline for EvalML?

Yes! EvalML allows you to create *custom pipelines* using modular components. This allows you to customize EvalML pipelines for your own needs or for AutoML.

### 4.11.10 Q: Does EvalML work with X algorithm?

EvalML is constantly improving and adding new components and will allow your own algorithms to be used as components in our pipelines.

CHAPTER

FIVE

# API REFERENCE

## 5.1 Demo Datasets

| | |
|---|---|
| *load_breast_cancer* | Load breast cancer dataset. Binary classification problem. |
| *load_diabetes* | Load diabetes dataset. Used for regression problem. |
| *load_fraud* | Load credit card fraud dataset. |
| *load_wine* | Load wine dataset. Multiclass problem. |
| *load_churn* | Load churn dataset, which can be used for binary classification problems. |

## 5.2 Preprocessing

Utilities to preprocess data before using evalml.

| | |
|---|---|
| *load_data* | Load features and target from file. |
| *target_distribution* | Get the target distributions. |
| *number_of_features* | Get the number of features of each specific dtype in a DataFrame. |
| *split_data* | Split data into train and test sets. |

## 5.3 Exceptions

| | |
|---|---|
| *MethodPropertyNotFoundError* | Exception to raise when a class is does not have an expected method or property. |
| *PipelineNotFoundError* | An exception raised when a particular pipeline is not found in automl search results. |
| *ObjectiveNotFoundError* | Exception to raise when specified objective does not exist. |
| *MissingComponentError* | An exception raised when a component is not found in all_components(). |
| *ComponentNotYetFittedError* | An exception to be raised when predict/predict_proba/transform is called on a component without fitting first. |
| *PipelineNotYetFittedError* | An exception to be raised when predict/predict_proba/transform is called on a pipeline without fitting first. |
| *AutoMLSearchException* | Exception raised when all pipelines in an automl batch return a score of NaN for the primary objective. |
| *PipelineScoreError* | An exception raised when a pipeline errors while scoring any objective in a list of objectives. |
| *DataCheckInitError* | Exception raised when a data check can't initialize with the parameters given. |
| *NullsInColumnWarning* | Warning thrown when there are null values in the column of interest. |

## 5.4 AutoML

### 5.4.1 AutoML Search Interface

| | |
|---|---|
| *AutoMLSearch* | Automated Pipeline search. |

### 5.4.2 AutoML Utils

| | |
|---|---|
| *search* | Given data and configuration, run an automl search. |
| *get_default_primary_search_objective* | Get the default primary search objective for a problem type. |
| *make_data_splitter* | Given the training data and ML problem parameters, compute a data splitting method to use during AutoML search. |
| *get_threshold_tuning_info* | Determine for a given automl config and pipeline what the threshold tuning objective should be and whether or not training data should be further split to achieve proper threshold tuning. |
| *resplit_training_data* | Further split the training data for a given pipeline. This is needed for binary pipelines in order to properly tune the threshold. |

### 5.4.3 AutoML Algorithm Classes

| | |
|---|---|
| *AutoMLAlgorithm* | Base class for the AutoML algorithms which power EvalML. |
| *IterativeAlgorithm* | An automl algorithm which first fits a base round of pipelines with default parameters, then does a round of parameter tuning on each pipeline in order of performance. |

### 5.4.4 AutoML Callbacks

| | |
|---|---|
| *silent_error_callback* | No-op. |
| *log_error_callback* | Logs the exception thrown as an error. |
| *raise_error_callback* | Raises the exception thrown by the AutoMLSearch object. |

### 5.4.5 AutoML Engines

| | |
|---|---|
| *SequentialEngine* | The default engine for the AutoML search. |
| *CFEngine* | The concurrent.futures (CF) engine. |
| *DaskEngine* | The dask engine. |

## 5.5 Pipelines

### 5.5.1 Pipeline Base Classes

| | |
|---|---|
| *PipelineBase* | Machine learning pipeline. |
| *ClassificationPipeline* | Pipeline subclass for all classification pipelines. |
| *BinaryClassificationPipeline* | Pipeline subclass for all binary classification pipelines. |
| *MulticlassClassificationPipeline* | Pipeline subclass for all multiclass classification pipelines. |
| *RegressionPipeline* | Pipeline subclass for all regression pipelines. |
| *TimeSeriesClassificationPipeline* | Pipeline base class for time series classification problems. |
| *TimeSeriesBinaryClassificationPipeline* | Pipeline base class for time series binary classification problems. |
| *TimeSeriesMulticlassClassificationPipeline* | Pipeline base class for time series multiclass classification problems. |
| *TimeSeriesRegressionPipeline* | Pipeline base class for time series regression problems. |

### 5.5.2 Pipeline Utils

| | |
|---|---|
| *make_pipeline* | Given input data, target data, an estimator class and the problem type, generates a pipeline class with a preprocessing chain which was recommended based on the inputs. The pipeline will be a subclass of the appropriate pipeline base class for the specified problem_type. |
| *generate_pipeline_code* | Creates and returns a string that contains the Python imports and code required for running the EvalML pipeline. |
| *rows_of_interest* | Get the row indices of the data that are closest to the threshold. Works only for binary classification problems and pipelines. |

## 5.6 Component Graphs

| | |
|---|---|
| *ComponentGraph* | Component graph for a pipeline as a directed acyclic graph (DAG). |

## 5.7 Components

### 5.7.1 Component Base Classes

Components represent a step in a pipeline.

| | |
|---|---|
| *ComponentBase* | Base class for all components. |
| *Transformer* | A component that may or may not need fitting that transforms data. These components are used before an estimator. |
| *Estimator* | A component that fits and predicts given data. |

### 5.7.2 Component Utils

| | |
|---|---|
| *allowed_model_families* | List the model types allowed for a particular problem type. |
| *get_estimators* | Returns the estimators allowed for a particular problem type. |
| *generate_component_code* | Creates and returns a string that contains the Python imports and code required for running the EvalML component. |

### 5.7.3 Transformers

Transformers are components that take in data as input and output transformed data.

| | |
|---|---|
| *DropColumns* | Drops specified columns in input data. |
| *SelectColumns* | Selects specified columns in input data. |
| *SelectByType* | Selects columns by specified Woodwork logical type or semantic tag in input data. |
| *OneHotEncoder* | A transformer that encodes categorical features in a one-hot numeric array. |
| *TargetEncoder* | A transformer that encodes categorical features into target encodings. |
| *PerColumnImputer* | Imputes missing data according to a specified imputation strategy per column. |
| *Imputer* | Imputes missing data according to a specified imputation strategy. |
| *SimpleImputer* | Imputes missing data according to a specified imputation strategy. Natural language columns are ignored. |
| *TimeSeriesImputer* | Imputes missing data according to a specified timeseries-specific imputation strategy. |
| *StandardScaler* | A transformer that standardizes input features by removing the mean and scaling to unit variance. |
| *RFRegressorSelectFromModel* | Selects top features based on importance weights using a Random Forest regressor. |
| *RFClassifierSelectFromModel* | Selects top features based on importance weights using a Random Forest classifier. |
| *RFClassifierRFESelector* | Selects relevant features using recursive feature elimination with a Random Forest Classifier. |
| *RFRegressorRFESelector* | Selects relevant features using recursive feature elimination with a Random Forest Regressor. |
| *DropNullColumns* | Transformer to drop features whose percentage of NaN values exceeds a specified threshold. |
| *DateTimeFeaturizer* | Transformer that can automatically extract features from datetime columns. |
| *NaturalLanguageFeaturizer* | Transformer that can automatically featurize text columns using featuretools' nlp_primitives. |
| *TimeSeriesFeaturizer* | Transformer that delays input features and target variable for time series problems. |
| *TimeSeriesRegularizer* | Transformer that regularizes an inconsistently spaced datetime column. |
| *DFSTransformer* | Featuretools DFS component that generates features for the input features. |
| *PolynomialDecomposer* | Removes trends and seasonality from time series by fitting a polynomial and moving average to the data. |
| *Undersampler* | Initializes an undersampling transformer to downsample the majority classes in the dataset. |
| *Oversampler* | SMOTE Oversampler component. Will automatically select whether to use SMOTE, SMOTEN, or SMOTENC based on inputs to the component. |

### 5.7.4 Estimators

#### Classifiers

Classifiers are components that output a predicted class label.

| | |
|---|---|
| *CatBoostClassifier* | CatBoost Classifier, a classifier that uses gradient-boosting on decision trees. CatBoost is an open-source library and natively supports categorical features. |
| *ElasticNetClassifier* | Elastic Net Classifier. Uses Logistic Regression with elasticnet penalty as the base estimator. |
| *ExtraTreesClassifier* | Extra Trees Classifier. |
| *RandomForestClassifier* | Random Forest Classifier. |
| *LightGBMClassifier* | LightGBM Classifier. |
| *LogisticRegressionClassifier* | Logistic Regression Classifier. |
| *XGBoostClassifier* | XGBoost Classifier. |
| *BaselineClassifier* | Classifier that predicts using the specified strategy. |
| *StackedEnsembleClassifier* | Stacked Ensemble Classifier. |
| *DecisionTreeClassifier* | Decision Tree Classifier. |
| *KNeighborsClassifier* | K-Nearest Neighbors Classifier. |
| *SVMClassifier* | Support Vector Machine Classifier. |
| *VowpalWabbitBinaryClassifier* | Vowpal Wabbit Binary Classifier. |
| *VowpalWabbitMulticlassClassifier* | Vowpal Wabbit Multiclass Classifier. |

#### Regressors

Regressors are components that output a predicted target value.

| | |
|---|---|
| *ARIMARegressor* | Autoregressive Integrated Moving Average Model. The three parameters (p, d, q) are the AR order, the degree of differencing, and the MA order. More information here: https://www.statsmodels.org/devel/generated/statsmodels.tsa.arima.model.ARIMA.html. |
| *CatBoostRegressor* | CatBoost Regressor, a regressor that uses gradient-boosting on decision trees. CatBoost is an open-source library and natively supports categorical features. |
| *ElasticNetRegressor* | Elastic Net Regressor. |
| *ExponentialSmoothingRegressor* | Holt-Winters Exponential Smoothing Forecaster. |
| *LinearRegressor* | Linear Regressor. |
| *ExtraTreesRegressor* | Extra Trees Regressor. |
| *RandomForestRegressor* | Random Forest Regressor. |
| *XGBoostRegressor* | XGBoost Regressor. |
| *BaselineRegressor* | Baseline regressor that uses a simple strategy to make predictions. This is useful as a simple baseline regressor to compare with other regressors. |
| *TimeSeriesBaselineEstimator* | Time series estimator that predicts using the naive forecasting approach. |
| *StackedEnsembleRegressor* | Stacked Ensemble Regressor. |
| *DecisionTreeRegressor* | Decision Tree Regressor. |
| *LightGBMRegressor* | LightGBM Regressor. |
| *SVMRegressor* | Support Vector Machine Regressor. |
| *VowpalWabbitRegressor* | Vowpal Wabbit Regressor. |

## 5.8 Model Understanding

### 5.8.1 Utility Methods

| | |
|---|---|
| *confusion_matrix* | Confusion matrix for binary and multiclass classification. |
| *normalize_confusion_matrix* | Normalizes a confusion matrix. |
| *precision_recall_curve* | Given labels and binary classifier predicted probabilities, compute and return the data representing a precision-recall curve. |
| *roc_curve* | Given labels and classifier predicted probabilities, compute and return the data representing a Receiver Operating Characteristic (ROC) curve. Works with binary or multiclass problems. |
| *calculate_permutation_importance* | Calculates permutation importance for features. |
| *calculate_permutation_importance_one_column* | Calculates permutation importance for one column in the original dataframe. |
| *binary_objective_vs_threshold* | Computes objective score as a function of potential binary classification decision thresholds for a fitted binary classification pipeline. |
| *get_prediction_vs_actual_over_time_data* | Get the data needed for the prediction_vs_actual_over_time plot. |
| *partial_dependence* | Calculates one or two-way partial dependence. |
| *get_prediction_vs_actual_data* | Combines y_true and y_pred into a single dataframe and adds a column for outliers. Used in *graph_prediction_vs_actual()*. |
| *get_linear_coefficients* | Returns a dataframe showing the features with the greatest predictive power for a linear model. |
| *t_sne* | Get the transformed output after fitting X to the embedded space using t-SNE. |
| *find_confusion_matrix_per_thresholds* | Gets the confusion matrix and histogram bins for each threshold as well as the best threshold per objective. Only works with Binary Classification Pipelines. |

### 5.8.2 Graph Utility Methods

| | |
|---|---|
| *graph_precision_recall_curve* | Generate and display a precision-recall plot. |
| *graph_roc_curve* | Generate and display a Receiver Operating Characteristic (ROC) plot for binary and multiclass classification problems. |
| *graph_confusion_matrix* | Generate and display a confusion matrix plot. |
| *graph_permutation_importance* | Generate a bar graph of the pipeline's permutation importance. |
| *graph_binary_objective_vs_threshold* | Generates a plot graphing objective score vs. decision thresholds for a fitted binary classification pipeline. |
| *graph_prediction_vs_actual* | Generate a scatter plot comparing the true and predicted values. Used for regression plotting. |
| *graph_prediction_vs_actual_over_time* | Plot the target values and predictions against time on the x-axis. |
| *graph_partial_dependence* | Create an one-way or two-way partial dependence plot. |
| *graph_t_sne* | Plot high dimensional data into lower dimensional space using t-SNE. |

### 5.8.3 Prediction Explanations

| | |
|---|---|
| *explain_predictions* | Creates a report summarizing the top contributing features for each data point in the input features. |
| *explain_predictions_best_worst* | Creates a report summarizing the top contributing features for the best and worst points in the dataset as measured by error to true labels. |

## 5.9 Objectives

### 5.9.1 Objective Base Classes

| | |
|---|---|
| *ObjectiveBase* | Base class for all objectives. |
| *BinaryClassificationObjective* | Base class for all binary classification objectives. |
| *MulticlassClassificationObjective* | Base class for all multiclass classification objectives. |
| *RegressionObjective* | Base class for all regression objectives. |

### 5.9.2 Domain-Specific Objectives

| | |
|---|---|
| *FraudCost* | Score the percentage of money lost of the total transaction amount process due to fraud. |
| *LeadScoring* | Lead scoring. |
| *CostBenefitMatrix* | Score using a cost-benefit matrix. Scores quantify the benefits of a given value, so greater numeric scores represents a better score. Costs and scores can be negative, indicating that a value is not beneficial. For example, in the case of monetary profit, a negative cost and/or score represents loss of cash flow. |

### 5.9.3 Classification Objectives

| | |
|---|---|
| *AccuracyBinary* | Accuracy score for binary classification. |
| *AccuracyMulticlass* | Accuracy score for multiclass classification. |
| *AUC* | AUC score for binary classification. |
| *AUCMacro* | AUC score for multiclass classification using macro averaging. |
| *AUCMicro* | AUC score for multiclass classification using micro averaging. |
| *AUCWeighted* | AUC Score for multiclass classification using weighted averaging. |
| *Gini* | Gini coefficient for binary classification. |
| *BalancedAccuracyBinary* | Balanced accuracy score for binary classification. |
| *BalancedAccuracyMulticlass* | Balanced accuracy score for multiclass classification. |
| *F1* | F1 score for binary classification. |
| *F1Micro* | F1 score for multiclass classification using micro averaging. |
| *F1Macro* | F1 score for multiclass classification using macro averaging. |
| *F1Weighted* | F1 score for multiclass classification using weighted averaging. |
| *LogLossBinary* | Log Loss for binary classification. |
| *LogLossMulticlass* | Log Loss for multiclass classification. |
| *MCCBinary* | Matthews correlation coefficient for binary classification. |
| *MCCMulticlass* | Matthews correlation coefficient for multiclass classification. |
| *Precision* | Precision score for binary classification. |
| *PrecisionMicro* | Precision score for multiclass classification using micro averaging. |
| *PrecisionMacro* | Precision score for multiclass classification using macro-averaging. |
| *PrecisionWeighted* | Precision score for multiclass classification using weighted averaging. |
| *Recall* | Recall score for binary classification. |
| *RecallMicro* | Recall score for multiclass classification using micro averaging. |
| *RecallMacro* | Recall score for multiclass classification using macro averaging. |
| *RecallWeighted* | Recall score for multiclass classification using weighted averaging. |

### 5.9.4 Regression Objectives

| | |
|---|---|
| *R2* | Coefficient of determination for regression. |
| *MAE* | Mean absolute error for regression. |
| *MAPE* | Mean absolute percentage error for time series regression. Scaled by 100 to return a percentage. |
| *MSE* | Mean squared error for regression. |
| *MeanSquaredLogError* | Mean squared log error for regression. |
| *MedianAE* | Median absolute error for regression. |
| *MaxError* | Maximum residual error for regression. |
| *ExpVariance* | Explained variance score for regression. |
| *RootMeanSquaredError* | Root mean squared error for regression. |
| *RootMeanSquaredLogError* | Root mean squared log error for regression. |

### 5.9.5 Objective Utils

| | |
|---|---|
| *get_all_objective_names* | Get a list of the names of all objectives. |
| *get_core_objectives* | Returns all core objective instances associated with the given problem type. |
| *get_core_objective_names* | Get a list of all valid core objectives. |
| *get_non_core_objectives* | Get non-core objective classes. |
| *get_objective* | Returns the Objective class corresponding to a given objective name. |
| *get_optimization_objectives* | Get objectives for optimization. |
| *get_ranking_objectives* | Get objectives for pipeline rankings. |
| *ranking_only_objectives* | Get ranking-only objective classes. |

## 5.10 Problem Types

| | |
|---|---|
| *handle_problem_types* | Handles problem_type by either returning the ProblemTypes or converting from a str. |
| *detect_problem_type* | Determine the type of problem is being solved based on the targets (binary vs multiclass classification, regression). Ignores missing and null data. |
| *ProblemTypes* | Enum defining the supported types of machine learning problems. |

## 5.11 Model Family

| | |
|---|---|
| *handle_model_family* | Handles model_family by either returning the ModelFamily or converting from a string. |
| *ModelFamily* | Enum for family of machine learning models. |

## 5.12 Tuners

| | |
|---|---|
| *Tuner* | Base Tuner class. |
| *SKOptTuner* | Bayesian Optimizer. |
| *GridSearchTuner* | Grid Search Optimizer, which generates all of the possible points to search for using a grid. |
| *RandomSearchTuner* | Random Search Optimizer. |

## 5.13 Data Checks

### 5.13.1 Data Check Classes

| | |
|---|---|
| *DataCheck* | Base class for all data checks. |
| *InvalidTargetDataCheck* | Check if the target data is considered invalid. |
| *NullDataCheck* | Check if there are any highly-null numerical, boolean, categorical, natural language, and unknown columns and rows in the input. |
| *IDColumnsDataCheck* | Check if any of the features are likely to be ID columns. |
| *TargetLeakageDataCheck* | Check if any of the features are highly correlated with the target by using mutual information, Pearson correlation, and other correlation metrics. |
| *OutliersDataCheck* | Checks if there are any outliers in input data by using IQR to determine score anomalies. |
| *NoVarianceDataCheck* | Check if the target or any of the features have no variance. |
| *ClassImbalanceDataCheck* | Check if any of the target labels are imbalanced, or if the number of values for each target are below 2 times the number of CV folds. Use for classification problems. |
| *MulticollinearityDataCheck* | Check if any set features are likely to be multicollinear. |
| *DateTimeFormatDataCheck* | Check if the datetime column has equally spaced intervals and is monotonically increasing or decreasing in order to be supported by time series estimators. |
| *TimeSeriesParametersDataCheck* | Checks whether the time series parameters are compatible with data splitting. |
| *TimeSeriesSplittingDataCheck* | Checks whether the time series target data is compatible with splitting. |
| *DataChecks* | A collection of data checks. |
| *DefaultDataChecks* | A collection of basic data checks that is used by AutoML by default. |

### 5.13.2 Data Check Messages

| | |
|---|---|
| *DataCheckMessage* | Base class for a message returned by a DataCheck, tagged by name. |
| *DataCheckError* | DataCheckMessage subclass for errors returned by data checks. |
| *DataCheckWarning* | DataCheckMessage subclass for warnings returned by data checks. |

### 5.13.3 Data Check Message Types

| | |
|---|---|
| *DataCheckMessageType* | Enum for type of data check message: WARNING or ERROR. |

### 5.13.4 Data Check Message Codes

| | |
|---|---|
| *DataCheckMessageCode* | Enum for data check message code. |

## 5.14 Utils

### 5.14.1 General Utils

| | |
|---|---|
| *import_or_raise* | Attempts to import the requested library by name. If the import fails, raises an ImportError or warning. |
| *convert_to_seconds* | Converts a string describing a length of time to its length in seconds. |
| *get_random_state* | Generates a numpy.random.RandomState instance using seed. |
| *get_random_seed* | Given a numpy.random.RandomState object, generate an int representing a seed value for another random number generator. Or, if given an int, return that int. |
| *pad_with_nans* | Pad the beginning num_to_pad rows with nans. |
| *drop_rows_with_nans* | Drop rows that have any NaNs in all dataframes or series. |
| *infer_feature_types* | Create a Woodwork structure from the given list, pandas, or numpy input, with specified types for columns. If a column's type is not specified, it will be inferred by Woodwork. |
| *save_plot* | Saves fig to filepath if specified, or to a default location if not. |
| *is_all_numeric* | Checks if the given DataFrame contains only numeric values. |
| *get_importable_subclasses* | Get importable subclasses of a base class. Used to list all of our estimators, transformers, components and pipelines dynamically. |

### Evalml

EvalML.

### Subpackages

### Automl

AutoMLSearch and related modules.

### Subpackages

### automl_algorithm

AutoML algorithms that power EvalML.

### Submodules

### automl_algorithm

Base class for the AutoML algorithms which power EvalML.

### Module Contents

### Classes Summary

| | |
|---|---|
| *AutoMLAlgorithm* | Base class for the AutoML algorithms which power EvalML. |

### Exceptions Summary

### Contents

**class** evalml.automl.automl_algorithm.automl_algorithm.**AutoMLAlgorithm**(*allowed_pipelines=None*, *search_parameters=None*, *tuner_class=None*, *text_in_ensembling=False*, *random_seed=0*, *n_jobs=- 1*)

Base class for the AutoML algorithms which power EvalML.

This class represents an automated machine learning (AutoML) algorithm. It encapsulates the decision-making logic behind an automl search, by both deciding which pipelines to evaluate next and by deciding what set of parameters to configure the pipeline with.

To use this interface, you must define a next_batch method which returns the next group of pipelines to evaluate on the training data. That method may access state and results recorded from the previous batches, although that information is not tracked in a general way in this base class. Overriding add_result is a convenient way to record pipeline evaluation info if necessary.

**Parameters**

- **allowed_pipelines** (*list(class)*) – A list of PipelineBase subclasses indicating the pipelines allowed in the search. The default of None indicates all pipelines for this problem type are allowed.
- **search_parameters** (*dict*) – Search parameter ranges specified for pipelines to iterate over.
- **tuner_class** (*class*) – A subclass of Tuner, to be used to find parameters for each pipeline. The default of None indicates the SKOptTuner will be used.
- **text_in_ensembling** (*boolean*) – If True and ensembling is True, then n_jobs will be set to 1 to avoid downstream sklearn stacking issues related to nltk. Defaults to None.
- **random_seed** (*int*) – Seed for the random number generator. Defaults to 0.

**Methods**

| | |
|---|---|
| *add_result* | Register results from evaluating a pipeline. |
| *batch_number* | Returns the number of batches which have been recommended so far. |
| *default_max_batches* | Returns the number of max batches AutoMLSearch should run by default. |
| *next_batch* | Get the next batch of pipelines to evaluate. |
| *num_pipelines_per_batch* | Return the number of pipelines in the nth batch. |
| *pipeline_number* | Returns the number of pipelines which have been recommended so far. |

**add_result**(*self*, *score_to_minimize*, *pipeline*, *trained_pipeline_results*)

Register results from evaluating a pipeline.

**Parameters**

- **score_to_minimize** (*float*) – The score obtained by this pipeline on the primary objective, converted so that lower values indicate better pipelines.
- **pipeline** (*PipelineBase*) – The trained pipeline object which was used to compute the score.
- **trained_pipeline_results** (*dict*) – Results from training a pipeline.

**Raises** **PipelineNotFoundError** – If pipeline is not allowed in search.

**property batch_number**(*self*)

Returns the number of batches which have been recommended so far.

**property default_max_batches**(*self*)

Returns the number of max batches AutoMLSearch should run by default.

**abstract next_batch**(*self*)

Get the next batch of pipelines to evaluate.

**Returns** A list of instances of PipelineBase subclasses, ready to be trained and evaluated.

**Return type** list[PipelineBase]

**abstract num_pipelines_per_batch**(*self*, *batch_number*)

Return the number of pipelines in the nth batch.

**Parameters** **batch_number** (*int*) – which batch to calculate the number of pipelines for.

**Returns** number of pipelines in the given batch.

**Return type** int

**property pipeline_number**(*self*)

Returns the number of pipelines which have been recommended so far.

**exception** evalml.automl.automl_algorithm.automl_algorithm.**AutoMLAlgorithmException**

Exception raised when an error is encountered during the computation of the automl algorithm.

## default_algorithm

An automl algorithm that consists of two modes: fast and long, where fast is a subset of long.

### Module Contents

### Classes Summary

| | |
|---|---|
| *DefaultAlgorithm* | An automl algorithm that consists of two modes: fast and long, where fast is a subset of long. |

### Contents

**class** evalml.automl.automl_algorithm.default_algorithm.**DefaultAlgorithm**(*X*, *y*, *problem_type*, *sampler_name*, *tuner_class=None*, *random_seed=0*, *search_parameters=None*, *n_jobs=1*, *text_in_ensembling=False*, *top_n=3*, *ensembling=False*, *num_long_explore_pipelines=50*, *num_long_pipelines_per_batch=10*, *allow_long_running_models=False*, *features=None*, *verbose=False*, *exclude_featurizers=None*)

An automl algorithm that consists of two modes: fast and long, where fast is a subset of long.

1. **Naive pipelines:**
   a. run baseline with default preprocessing pipeline
   b. run naive linear model with default preprocessing pipeline
   c. run basic RF pipeline with default preprocessing pipeline

2. **Naive pipelines with feature selection**
    a. subsequent pipelines will use the selected features with a SelectedColumns transformer

At this point we have a single pipeline candidate for preprocessing and feature selection

3. **Pipelines with preprocessing components:**
    a. scan rest of estimators (our current batch 1).
4. First ensembling run

Fast mode ends here. Begin long mode.

6. **Run top 3 estimators:**
    a. Generate 50 random parameter sets. Run all 150 in one batch
7. Second ensembling run
8. **Repeat these indefinitely until stopping criterion is met:**
    a. For each of the previous top 3 estimators, sample 10 parameters from the tuner. Run all 30 in one batch
    b. Run ensembling

**Parameters**

- **X** (*pd.DataFrame*) – Training data.
- **y** (*pd.Series*) – Target data.
- **problem_type** (*ProblemType*) – Problem type associated with training data.
- **sampler_name** (*BaseSampler*) – Sampler to use for preprocessing.
- **tuner_class** (*class*) – A subclass of Tuner, to be used to find parameters for each pipeline. The default of None indicates the SKOptTuner will be used.
- **random_seed** (*int*) – Seed for the random number generator. Defaults to 0.
- **search_parameters** (*dict or None*) – Pipeline-level parameters and custom hyperparameter ranges specified for pipelines to iterate over. Hyperparameter ranges must be passed in as skopt.space objects. Defaults to None.
- **n_jobs** (*int or None*) – Non-negative integer describing level of parallelism used for pipelines. Defaults to -1.
- **text_in_ensembling** (*boolean*) – If True and ensembling is True, then n_jobs will be set to 1 to avoid downstream sklearn stacking issues related to nltk. Defaults to False.
- **top_n** (*int*) – top n number of pipelines to use for long mode.
- **num_long_explore_pipelines** (*int*) – number of pipelines to explore for each top n pipeline at the start of long mode.
- **num_long_pipelines_per_batch** (*int*) – number of pipelines per batch for each top n pipeline through long mode.
- **allow_long_running_models** (*bool*) – Whether or not to allow longer-running models for large multiclass problems. If False and no pipelines, component graphs, or model families are provided, AutoMLSearch will not use Elastic Net or XGBoost when there are more than 75 multiclass targets and will not use CatBoost when there are more than 150 multiclass targets. Defaults to False.

- **features** (*list*) – List of features to run DFS on in AutoML pipelines. Defaults to None. Features will only be computed if the columns used by the feature exist in the input and if the feature has not been computed yet.
- **verbose** (*boolean*) – Whether or not to display logging information regarding pipeline building. Defaults to False.
- **exclude_featurizers** (*list[str]*) – A list of featurizer components to exclude from the pipelines built by DefaultAlgorithm. Valid options are "DatetimeFeaturizer", "EmailFeaturizer", "URLFeaturizer", "NaturalLanguageFeaturizer", "TimeSeriesFeaturizer"

**Methods**

| | |
|---|---|
| *add_result* | Register results from evaluating a pipeline. In batch number 2, the selected column names from the feature selector are taken to be used in a column selector. Information regarding the best pipeline is updated here as well. |
| *batch_number* | Returns the number of batches which have been recommended so far. |
| *default_max_batches* | Returns the number of max batches AutoMLSearch should run by default. |
| *next_batch* | Get the next batch of pipelines to evaluate. |
| *num_pipelines_per_batch* | Return the number of pipelines in the nth batch. |
| *pipeline_number* | Returns the number of pipelines which have been recommended so far. |

**add_result**(*self*, *score_to_minimize*, *pipeline*, *trained_pipeline_results*, *cached_data=None*)

Register results from evaluating a pipeline. In batch number 2, the selected column names from the feature selector are taken to be used in a column selector. Information regarding the best pipeline is updated here as well.

**Parameters**

- **score_to_minimize** (*float*) – The score obtained by this pipeline on the primary objective, converted so that lower values indicate better pipelines.
- **pipeline** (*PipelineBase*) – The trained pipeline object which was used to compute the score.
- **trained_pipeline_results** (*dict*) – Results from training a pipeline.
- **cached_data** (*dict*) – A dictionary of cached data, where the keys are the model family. Expected to be of format {model_family: {hash1: trained_component_graph, hash2: trained_component_graph. . . }. . . }. Defaults to None.

**property batch_number**(*self*)

Returns the number of batches which have been recommended so far.

**property default_max_batches**(*self*)

Returns the number of max batches AutoMLSearch should run by default.

**next_batch**(*self*)

Get the next batch of pipelines to evaluate.

**Returns** a list of instances of PipelineBase subclasses, ready to be trained and evaluated.

**Return type** list(PipelineBase)

**num_pipelines_per_batch**(*self*, *batch_number*)

Return the number of pipelines in the nth batch.

**Parameters** **batch_number** (*int*) – which batch to calculate the number of pipelines for.

**Returns** number of pipelines in the given batch.

**Return type** int

**property pipeline_number**(*self*)

Returns the number of pipelines which have been recommended so far.

### iterative_algorithm

An automl algorithm which first fits a base round of pipelines with default parameters, then does a round of parameter tuning on each pipeline in order of performance.

#### Module Contents

#### Classes Summary

| | |
|---|---|
| *IterativeAlgorithm* | An automl algorithm which first fits a base round of pipelines with default parameters, then does a round of parameter tuning on each pipeline in order of performance. |

#### Contents

**class** evalml.automl.automl_algorithm.iterative_algorithm.**IterativeAlgorithm**(*X*, *y*, *problem_type*, *sampler_name=None*, *allowed_model_families=None*, *allowed_component_graphs=None*, *max_batches=None*, *max_iterations=None*, *tuner_class=None*, *random_seed=0*, *pipelines_per_batch=5*, *n_jobs=- 1*, *number_features=None*, *ensembling=False*, *text_in_ensembling=False*, *search_parameters=None*, *_estimator_family_order=None*, *allow_long_running_models=False*, *features=None*, *verbose=False*, *exclude_featurizers=None*)

An automl algorithm which first fits a base round of pipelines with default parameters, then does a round of parameter tuning on each pipeline in order of performance.

**Parameters**

- **X** (*pd.DataFrame*) – Training data.
- **y** (*pd.Series*) – Target data.
- **problem_type** (*ProblemType*) – Problem type associated with training data.
- **sampler_name** (*BaseSampler*) – Sampler to use for preprocessing. Defaults to None.
- **allowed_model_families** (*list(str, ModelFamily)*) – The model families to search. The default of None searches over all model families. Run evalml.pipelines.components.utils.allowed_model_families("binary") to see options. Change *binary* to *multiclass* or *regression* depending on the problem type. Note that if allowed_pipelines is provided, this parameter will be ignored.
- **allowed_component_graphs** (*dict*) – A dictionary of lists or ComponentGraphs indicating the component graphs allowed in the search. The format should follow { "Name_0": [list_of_components], "Name_1": [ComponentGraph(...)] }

  The default of None indicates all pipeline component graphs for this problem type are allowed. Setting this field will cause allowed_model_families to be ignored.

  e.g. allowed_component_graphs = { "My_Graph": ["Imputer", "One Hot Encoder", "Random Forest Classifier"] }
- **max_batches** (*int*) – The maximum number of batches to be evaluated. Used to determine ensembling. Defaults to None.

- **max_iterations** (*int*) – The maximum number of iterations to be evaluated. Used to determine ensembling. Defaults to None.
- **tuner_class** (*class*) – A subclass of Tuner, to be used to find parameters for each pipeline. The default of None indicates the SKOptTuner will be used.
- **random_seed** (*int*) – Seed for the random number generator. Defaults to 0.
- **pipelines_per_batch** (*int*) – The number of pipelines to be evaluated in each batch, after the first batch. Defaults to 5.
- **n_jobs** (*int or None*) – Non-negative integer describing level of parallelism used for pipelines. Defaults to None.
- **number_features** (*int*) – The number of columns in the input features. Defaults to None.
- **ensembling** (*boolean*) – If True, runs ensembling in a separate batch after every allowed pipeline class has been iterated over. Defaults to False.
- **text_in_ensembling** (*boolean*) – If True and ensembling is True, then n_jobs will be set to 1 to avoid downstream sklearn stacking issues related to nltk. Defaults to False.
- **search_parameters** (*dict or None*) – Pipeline-level parameters and custom hyperparameter ranges specified for pipelines to iterate over. Hyperparameter ranges must be passed in as skopt.space objects. Defaults to None.
- **_estimator_family_order** (*list(ModelFamily) or None*) – specify the sort order for the first batch. Defaults to None, which uses _ESTIMATOR_FAMILY_ORDER.
- **allow_long_running_models** (*bool*) – Whether or not to allow longer-running models for large multiclass problems. If False and no pipelines, component graphs, or model families are provided, AutoMLSearch will not use Elastic Net or XGBoost when there are more than 75 multiclass targets and will not use CatBoost when there are more than 150 multiclass targets. Defaults to False.
- **features** (*list*) – List of features to run DFS on in AutoML pipelines. Defaults to None. Features will only be computed if the columns used by the feature exist in the input and if the feature itself is not in input.
- **verbose** (*boolean*) – Whether or not to display logging information regarding pipeline building. Defaults to False.
- **exclude_featurizers** (*list[str]*) – A list of featurizer components to exclude from the pipelines built by IterativeAlgorithm. Valid options are "DatetimeFeaturizer", "EmailFeaturizer", "URLFeaturizer", "NaturalLanguageFeaturizer", "TimeSeriesFeaturizer"

**Methods**

| | |
|---|---|
| *add_result* | Register results from evaluating a pipeline. |
| *batch_number* | Returns the number of batches which have been recommended so far. |
| *default_max_batches* | Returns the number of max batches AutoMLSearch should run by default. |
| *next_batch* | Get the next batch of pipelines to evaluate. |
| *num_pipelines_per_batch* | Return the number of pipelines in the nth batch. |
| *pipeline_number* | Returns the number of pipelines which have been recommended so far. |

**add_result**(*self*, *score_to_minimize*, *pipeline*, *trained_pipeline_results*, *cached_data=None*)

Register results from evaluating a pipeline.

**Parameters**

- **score_to_minimize** (*float*) – The score obtained by this pipeline on the primary objective, converted so that lower values indicate better pipelines.
- **pipeline** (*PipelineBase*) – The trained pipeline object which was used to compute the score.
- **trained_pipeline_results** (*dict*) – Results from training a pipeline.
- **cached_data** (*dict*) – A dictionary of cached data, where the keys are the model family. Expected to be of format {model_family: {hash1: trained_component_graph, hash2: trained_component_graph…}…}. Defaults to None.

**Raises** **ValueError** – If default parameters are not in the acceptable hyperparameter ranges.

**property batch_number**(*self*)

Returns the number of batches which have been recommended so far.

**property default_max_batches**(*self*)

Returns the number of max batches AutoMLSearch should run by default.

**next_batch**(*self*)

Get the next batch of pipelines to evaluate.

**Returns** A list of instances of PipelineBase subclasses, ready to be trained and evaluated.

**Return type** list[PipelineBase]

**Raises** **AutoMLAlgorithmException** – If no results were reported from the first batch.

**num_pipelines_per_batch**(*self*, *batch_number*)

Return the number of pipelines in the nth batch.

**Parameters** **batch_number** (*int*) – which batch to calculate the number of pipelines for.

**Returns** number of pipelines in the given batch.

**Return type** int

**property pipeline_number**(*self*)

Returns the number of pipelines which have been recommended so far.

## Package Contents

## Classes Summary

| | |
|---|---|
| *AutoMLAlgorithm* | Base class for the AutoML algorithms which power EvalML. |
| *DefaultAlgorithm* | An automl algorithm that consists of two modes: fast and long, where fast is a subset of long. |
| *IterativeAlgorithm* | An automl algorithm which first fits a base round of pipelines with default parameters, then does a round of parameter tuning on each pipeline in order of performance. |

### Exceptions Summary

### Contents

**class** evalml.automl.automl_algorithm.**AutoMLAlgorithm**(*allowed_pipelines=None*, *search_parameters=None*, *tuner_class=None*, *text_in_ensembling=False*, *random_seed=0*, *n_jobs=- 1*)

Base class for the AutoML algorithms which power EvalML.

This class represents an automated machine learning (AutoML) algorithm. It encapsulates the decision-making logic behind an automl search, by both deciding which pipelines to evaluate next and by deciding what set of parameters to configure the pipeline with.

To use this interface, you must define a next_batch method which returns the next group of pipelines to evaluate on the training data. That method may access state and results recorded from the previous batches, although that information is not tracked in a general way in this base class. Overriding add_result is a convenient way to record pipeline evaluation info if necessary.

**Parameters**

- **allowed_pipelines** (*list(class)*) – A list of PipelineBase subclasses indicating the pipelines allowed in the search. The default of None indicates all pipelines for this problem type are allowed.
- **search_parameters** (*dict*) – Search parameter ranges specified for pipelines to iterate over.
- **tuner_class** (*class*) – A subclass of Tuner, to be used to find parameters for each pipeline. The default of None indicates the SKOptTuner will be used.
- **text_in_ensembling** (*boolean*) – If True and ensembling is True, then n_jobs will be set to 1 to avoid downstream sklearn stacking issues related to nltk. Defaults to None.
- **random_seed** (*int*) – Seed for the random number generator. Defaults to 0.

**Methods**

| | |
|---|---|
| *add_result* | Register results from evaluating a pipeline. |
| *batch_number* | Returns the number of batches which have been recommended so far. |
| *default_max_batches* | Returns the number of max batches AutoMLSearch should run by default. |
| *next_batch* | Get the next batch of pipelines to evaluate. |
| *num_pipelines_per_batch* | Return the number of pipelines in the nth batch. |
| *pipeline_number* | Returns the number of pipelines which have been recommended so far. |

**add_result**(*self*, *score_to_minimize*, *pipeline*, *trained_pipeline_results*)

Register results from evaluating a pipeline.

**Parameters**

- **score_to_minimize** (*float*) – The score obtained by this pipeline on the primary objective, converted so that lower values indicate better pipelines.

- **pipeline** (*PipelineBase*) – The trained pipeline object which was used to compute the score.
- **trained_pipeline_results** (*dict*) – Results from training a pipeline.

**Raises** **PipelineNotFoundError** – If pipeline is not allowed in search.

**property batch_number**(*self*)

Returns the number of batches which have been recommended so far.

**property default_max_batches**(*self*)

Returns the number of max batches AutoMLSearch should run by default.

**abstract next_batch**(*self*)

Get the next batch of pipelines to evaluate.

**Returns** A list of instances of PipelineBase subclasses, ready to be trained and evaluated.

**Return type** list[PipelineBase]

**abstract num_pipelines_per_batch**(*self*, *batch_number*)

Return the number of pipelines in the nth batch.

**Parameters** **batch_number** (*int*) – which batch to calculate the number of pipelines for.

**Returns** number of pipelines in the given batch.

**Return type** int

**property pipeline_number**(*self*)

Returns the number of pipelines which have been recommended so far.

**exception** evalml.automl.automl_algorithm.**AutoMLAlgorithmException**

Exception raised when an error is encountered during the computation of the automl algorithm.

**class** evalml.automl.automl_algorithm.**DefaultAlgorithm**(*X*, *y*, *problem_type*, *sampler_name*, *tuner_class=None*, *random_seed=0*, *search_parameters=None*, *n_jobs=1*, *text_in_ensembling=False*, *top_n=3*, *ensembling=False*, *num_long_explore_pipelines=50*, *num_long_pipelines_per_batch=10*, *allow_long_running_models=False*, *features=None*, *verbose=False*, *exclude_featurizers=None*)

An automl algorithm that consists of two modes: fast and long, where fast is a subset of long.

1. **Naive pipelines:**
   a. run baseline with default preprocessing pipeline
   b. run naive linear model with default preprocessing pipeline
   c. run basic RF pipeline with default preprocessing pipeline
2. **Naive pipelines with feature selection**
   a. subsequent pipelines will use the selected features with a SelectedColumns transformer

At this point we have a single pipeline candidate for preprocessing and feature selection

3. **Pipelines with preprocessing components:**
   a. scan rest of estimators (our current batch 1).

4. First ensembling run

Fast mode ends here. Begin long mode.

6. **Run top 3 estimators:**
    a. Generate 50 random parameter sets. Run all 150 in one batch
7. Second ensembling run
8. **Repeat these indefinitely until stopping criterion is met:**
    a. For each of the previous top 3 estimators, sample 10 parameters from the tuner. Run all 30 in one batch
    b. Run ensembling

**Parameters**

- **X** (*pd.DataFrame*) – Training data.
- **y** (*pd.Series*) – Target data.
- **problem_type** (*ProblemType*) – Problem type associated with training data.
- **sampler_name** (*BaseSampler*) – Sampler to use for preprocessing.
- **tuner_class** (*class*) – A subclass of Tuner, to be used to find parameters for each pipeline. The default of None indicates the SKOptTuner will be used.
- **random_seed** (*int*) – Seed for the random number generator. Defaults to 0.
- **search_parameters** (*dict or None*) – Pipeline-level parameters and custom hyperparameter ranges specified for pipelines to iterate over. Hyperparameter ranges must be passed in as skopt.space objects. Defaults to None.
- **n_jobs** (*int or None*) – Non-negative integer describing level of parallelism used for pipelines. Defaults to -1.
- **text_in_ensembling** (*boolean*) – If True and ensembling is True, then n_jobs will be set to 1 to avoid downstream sklearn stacking issues related to nltk. Defaults to False.
- **top_n** (*int*) – top n number of pipelines to use for long mode.
- **num_long_explore_pipelines** (*int*) – number of pipelines to explore for each top n pipeline at the start of long mode.
- **num_long_pipelines_per_batch** (*int*) – number of pipelines per batch for each top n pipeline through long mode.
- **allow_long_running_models** (*bool*) – Whether or not to allow longer-running models for large multiclass problems. If False and no pipelines, component graphs, or model families are provided, AutoMLSearch will not use Elastic Net or XGBoost when there are more than 75 multiclass targets and will not use CatBoost when there are more than 150 multiclass targets. Defaults to False.
- **features** (*list*) – List of features to run DFS on in AutoML pipelines. Defaults to None. Features will only be computed if the columns used by the feature exist in the input and if the feature has not been computed yet.
- **verbose** (*boolean*) – Whether or not to display logging information regarding pipeline building. Defaults to False.

- **exclude_featurizers** (*list[str]*) – A list of featurizer components to exclude from the pipelines built by DefaultAlgorithm. Valid options are "DatetimeFeaturizer", "EmailFeaturizer", "URLFeaturizer", "NaturalLanguageFeaturizer", "TimeSeriesFeaturizer"

**Methods**

| | |
|---|---|
| *add_result* | Register results from evaluating a pipeline. In batch number 2, the selected column names from the feature selector are taken to be used in a column selector. Information regarding the best pipeline is updated here as well. |
| *batch_number* | Returns the number of batches which have been recommended so far. |
| *default_max_batches* | Returns the number of max batches AutoMLSearch should run by default. |
| *next_batch* | Get the next batch of pipelines to evaluate. |
| *num_pipelines_per_batch* | Return the number of pipelines in the nth batch. |
| *pipeline_number* | Returns the number of pipelines which have been recommended so far. |

**add_result**(*self*, *score_to_minimize*, *pipeline*, *trained_pipeline_results*, *cached_data=None*)

Register results from evaluating a pipeline. In batch number 2, the selected column names from the feature selector are taken to be used in a column selector. Information regarding the best pipeline is updated here as well.

**Parameters**

- **score_to_minimize** (*float*) – The score obtained by this pipeline on the primary objective, converted so that lower values indicate better pipelines.
- **pipeline** (*PipelineBase*) – The trained pipeline object which was used to compute the score.
- **trained_pipeline_results** (*dict*) – Results from training a pipeline.
- **cached_data** (*dict*) – A dictionary of cached data, where the keys are the model family. Expected to be of format {model_family: {hash1: trained_component_graph, hash2: trained_component_graph. . . }. . . }. Defaults to None.

**property batch_number**(*self*)

Returns the number of batches which have been recommended so far.

**property default_max_batches**(*self*)

Returns the number of max batches AutoMLSearch should run by default.

**next_batch**(*self*)

Get the next batch of pipelines to evaluate.

**Returns** a list of instances of PipelineBase subclasses, ready to be trained and evaluated.

**Return type** list(PipelineBase)

**num_pipelines_per_batch**(*self*, *batch_number*)

Return the number of pipelines in the nth batch.

**Parameters** **batch_number** (*int*) – which batch to calculate the number of pipelines for.

**Returns** number of pipelines in the given batch.

**Return type** int

**property pipeline_number**(*self*)

Returns the number of pipelines which have been recommended so far.

**class** evalml.automl.automl_algorithm.**IterativeAlgorithm**(*X*, *y*, *problem_type*, *sampler_name=None*, *allowed_model_families=None*, *allowed_component_graphs=None*, *max_batches=None*, *max_iterations=None*, *tuner_class=None*, *random_seed=0*, *pipelines_per_batch=5*, *n_jobs=- 1*, *number_features=None*, *ensembling=False*, *text_in_ensembling=False*, *search_parameters=None*, *_estimator_family_order=None*, *allow_long_running_models=False*, *features=None*, *verbose=False*, *exclude_featurizers=None*)

An automl algorithm which first fits a base round of pipelines with default parameters, then does a round of parameter tuning on each pipeline in order of performance.

**Parameters**

- **X** (*pd.DataFrame*) – Training data.
- **y** (*pd.Series*) – Target data.
- **problem_type** (*ProblemType*) – Problem type associated with training data.
- **sampler_name** (*BaseSampler*) – Sampler to use for preprocessing. Defaults to None.
- **allowed_model_families** (*list(str, ModelFamily)*) – The model families to search. The default of None searches over all model families. Run evalml.pipelines.components.utils.allowed_model_families("binary") to see options. Change *binary* to *multiclass* or *regression* depending on the problem type. Note that if allowed_pipelines is provided, this parameter will be ignored.
- **allowed_component_graphs** (*dict*) – A dictionary of lists or ComponentGraphs indicating the component graphs allowed in the search. The format should follow { "Name_0": [list_of_components], "Name_1": [ComponentGraph(...)] }

  The default of None indicates all pipeline component graphs for this problem type are allowed. Setting this field will cause allowed_model_families to be ignored.

  e.g. allowed_component_graphs = { "My_Graph": ["Imputer", "One Hot Encoder", "Random Forest Classifier"] }
- **max_batches** (*int*) – The maximum number of batches to be evaluated. Used to determine ensembling. Defaults to None.
- **max_iterations** (*int*) – The maximum number of iterations to be evaluated. Used to determine ensembling. Defaults to None.
- **tuner_class** (*class*) – A subclass of Tuner, to be used to find parameters for each pipeline. The default of None indicates the SKOptTuner will be used.
- **random_seed** (*int*) – Seed for the random number generator. Defaults to 0.
- **pipelines_per_batch** (*int*) – The number of pipelines to be evaluated in each batch, after the first batch. Defaults to 5.

- **n_jobs** (*int or None*) – Non-negative integer describing level of parallelism used for pipelines. Defaults to None.
- **number_features** (*int*) – The number of columns in the input features. Defaults to None.
- **ensembling** (*boolean*) – If True, runs ensembling in a separate batch after every allowed pipeline class has been iterated over. Defaults to False.
- **text_in_ensembling** (*boolean*) – If True and ensembling is True, then n_jobs will be set to 1 to avoid downstream sklearn stacking issues related to nltk. Defaults to False.
- **search_parameters** (*dict or None*) – Pipeline-level parameters and custom hyperparameter ranges specified for pipelines to iterate over. Hyperparameter ranges must be passed in as skopt.space objects. Defaults to None.
- **_estimator_family_order** (*list(ModelFamily) or None*) – specify the sort order for the first batch. Defaults to None, which uses _ESTIMATOR_FAMILY_ORDER.
- **allow_long_running_models** (*bool*) – Whether or not to allow longer-running models for large multiclass problems. If False and no pipelines, component graphs, or model families are provided, AutoMLSearch will not use Elastic Net or XGBoost when there are more than 75 multiclass targets and will not use CatBoost when there are more than 150 multiclass targets. Defaults to False.
- **features** (*list*) – List of features to run DFS on in AutoML pipelines. Defaults to None. Features will only be computed if the columns used by the feature exist in the input and if the feature itself is not in input.
- **verbose** (*boolean*) – Whether or not to display logging information regarding pipeline building. Defaults to False.
- **exclude_featurizers** (*list[str]*) – A list of featurizer components to exclude from the pipelines built by IterativeAlgorithm. Valid options are "DatetimeFeaturizer", "EmailFeaturizer", "URLFeaturizer", "NaturalLanguageFeaturizer", "TimeSeriesFeaturizer"

**Methods**

| | |
|---|---|
| *add_result* | Register results from evaluating a pipeline. |
| *batch_number* | Returns the number of batches which have been recommended so far. |
| *default_max_batches* | Returns the number of max batches AutoMLSearch should run by default. |
| *next_batch* | Get the next batch of pipelines to evaluate. |
| *num_pipelines_per_batch* | Return the number of pipelines in the nth batch. |
| *pipeline_number* | Returns the number of pipelines which have been recommended so far. |

**add_result**(*self*, *score_to_minimize*, *pipeline*, *trained_pipeline_results*, *cached_data=None*)

Register results from evaluating a pipeline.

**Parameters**

- **score_to_minimize** (*float*) – The score obtained by this pipeline on the primary objective, converted so that lower values indicate better pipelines.
- **pipeline** (*PipelineBase*) – The trained pipeline object which was used to compute the score.
- **trained_pipeline_results** (*dict*) – Results from training a pipeline.

- **cached_data** (*dict*) – A dictionary of cached data, where the keys are the model family. Expected to be of format {model_family: {hash1: trained_component_graph, hash2: trained_component_graph…}…}. Defaults to None.

**Raises** **ValueError** – If default parameters are not in the acceptable hyperparameter ranges.

**property batch_number**(*self*)

Returns the number of batches which have been recommended so far.

**property default_max_batches**(*self*)

Returns the number of max batches AutoMLSearch should run by default.

**next_batch**(*self*)

Get the next batch of pipelines to evaluate.

**Returns** A list of instances of PipelineBase subclasses, ready to be trained and evaluated.

**Return type** list[PipelineBase]

**Raises** *AutoMLAlgorithmException* – If no results were reported from the first batch.

**num_pipelines_per_batch**(*self*, *batch_number*)

Return the number of pipelines in the nth batch.

**Parameters** **batch_number** (*int*) – which batch to calculate the number of pipelines for.

**Returns** number of pipelines in the given batch.

**Return type** int

**property pipeline_number**(*self*)

Returns the number of pipelines which have been recommended so far.

## engine

EvalML Engine classes used to evaluate pipelines in AutoMLSearch.

### Submodules

### cf_engine

Custom CFClient API to match Dask's CFClient and allow context management.

### Module Contents

### Classes Summary

| | |
|---|---|
| *CFClient* | Custom CFClient API to match Dask's CFClient and allow context management. |
| *CFComputation* | A Future-like wrapper around jobs created by the CFEngine. |
| *CFEngine* | The concurrent.futures (CF) engine. |

### Contents

**class** evalml.automl.engine.cf_engine.**CFClient**(*pool*)

Custom CFClient API to match Dask's CFClient and allow context management.

**Parameters** **pool** (*cf.ThreadPoolExecutor or cf.ProcessPoolExecutor*) – The resource pool to execute the futures work on.

**Methods**

| | |
|---|---|
| *close* | Closes the underlying Executor. |
| *is_closed* | Property that determines whether the Engine's Client's resources are closed. |
| *submit* | Pass through to imitate Dask's Client API. |

**close**(*self*)

Closes the underlying Executor.

**property is_closed**(*self*)

Property that determines whether the Engine's Client's resources are closed.

**submit**(*self*, **args*, ***kwargs*)

Pass through to imitate Dask's Client API.

**class** evalml.automl.engine.cf_engine.**CFComputation**(*future*)

A Future-like wrapper around jobs created by the CFEngine.

**Parameters** **future** (*cf.Future*) – The concurrent.futures.Future that is desired to be executed.

**Methods**

| | |
|---|---|
| *cancel* | Cancel the current computation. |
| *done* | Returns whether the computation is done. |
| *get_result* | Gets the computation result. Will block until the computation is finished. |
| *is_cancelled* | Returns whether computation was cancelled. |

**cancel**(*self*)

Cancel the current computation.

**Returns**

**False if the call is currently being executed or finished running** and cannot be cancelled. True if the call can be canceled.

**Return type** bool

**done**(*self*)

Returns whether the computation is done.

**get_result**(*self*)

Gets the computation result. Will block until the computation is finished.

**Raises**

- **Exception** – If computation fails. Returns traceback.
- **cf.TimeoutError** – If computation takes longer than default timeout time.

- **cf.CancelledError** – If computation was canceled before completing.

**Returns** The result of the requested job.

**property is_cancelled**(*self*)

Returns whether computation was cancelled.

**class** evalml.automl.engine.cf_engine.**CFEngine**(*client=None*)

The concurrent.futures (CF) engine.

**Parameters client** (*None or CFClient*) – If None, creates a threaded pool for processing. Defaults to None.

**Methods**

| | |
|---|---|
| *close* | Function to properly shutdown the Engine's Client's resources. |
| *is_closed* | Property that determines whether the Engine's Client's resources are shutdown. |
| *setup_job_log* | Set up logger for job. |
| *submit_evaluation_job* | Send evaluation job to cluster. |
| *submit_scoring_job* | Send scoring job to cluster. |
| *submit_training_job* | Send training job to cluster. |

**close**(*self*)

Function to properly shutdown the Engine's Client's resources.

**property is_closed**(*self*)

Property that determines whether the Engine's Client's resources are shutdown.

**static setup_job_log**()

Set up logger for job.

**submit_evaluation_job**(*self, automl_config, pipeline, X, y, X_holdout=None, y_holdout=None*)

Send evaluation job to cluster.

**Parameters**

- **automl_config** – Structure containing data passed from AutoMLSearch instance.
- **pipeline** (*pipeline.PipelineBase*) – Pipeline to evaluate.
- **X** (*pd.DataFrame*) – Input data for modeling.
- **y** (*pd.Series*) – Target data for modeling.
- **X_holdout** (*pd.Series*) – Holdout input data for holdout scoring.
- **y_holdout** (*pd.Series*) – Holdout target data for holdout scoring.

**Returns**

**An object wrapping a reference to a future-like computation** occurring in the resource pool

**Return type** *CFComputation*

**submit_scoring_job**(*self, automl_config, pipeline, X, y, objectives, X_train=None, y_train=None*)

Send scoring job to cluster.

**Parameters**

- **automl_config** – Structure containing data passed from AutoMLSearch instance.
- **pipeline** (*pipeline.PipelineBase*) – Pipeline to train.
- **X** (*pd.DataFrame*) – Input data for modeling.
- **y** (*pd.Series*) – Target data for modeling.
- **X_train** (*pd.DataFrame*) – Training features. Used for feature engineering in time series.
- **y_train** (*pd.Series*) – Training target. Used for feature engineering in time series.
- **objectives** (*list[ObjectiveBase]*) – Objectives to score on.

**Returns**

**An object wrapping a reference to a future-like computation** occurring in the resource pool.

**Return type** *CFComputation*

**submit_training_job**(*self*, *automl_config*, *pipeline*, *X*, *y*)

Send training job to cluster.

**Parameters**

- **automl_config** – Structure containing data passed from AutoMLSearch instance.
- **pipeline** (*pipeline.PipelineBase*) – Pipeline to train.
- **X** (*pd.DataFrame*) – Input data for modeling.
- **y** (*pd.Series*) – Target data for modeling.

**Returns**

**An object wrapping a reference to a future-like computation** occurring in the resource pool

**Return type** *CFComputation*

### dask_engine

A Future-like wrapper around jobs created by the DaskEngine.

### Module Contents

### Classes Summary

| | |
|---|---|
| *DaskComputation* | A Future-like wrapper around jobs created by the DaskEngine. |
| *DaskEngine* | The dask engine. |

### Contents

**class** evalml.automl.engine.dask_engine.**DaskComputation**(*dask_future*)

A Future-like wrapper around jobs created by the DaskEngine.

**Parameters** **dask_future** (*callable*) – Computation to do.

**Methods**

| | |
|---|---|
| *cancel* | Cancel the current computation. |
| *done* | Returns whether the computation is done. |
| *get_result* | Gets the computation result. Will block until the computation is finished. |
| *is_cancelled* | Returns whether computation was cancelled. |

**cancel**(*self*)

Cancel the current computation.

**done**(*self*)

Returns whether the computation is done.

**get_result**(*self*)

Gets the computation result. Will block until the computation is finished.

**Raises** **Exception** – If computation fails. Returns traceback.

**Returns** Computation results.

**property is_cancelled**(*self*)

Returns whether computation was cancelled.

**class** evalml.automl.engine.dask_engine.**DaskEngine**(*cluster=None*)

The dask engine.

**Parameters** **cluster** (*None or dd.Client*) – If None, creates a local, threaded Dask client for processing. Defaults to None.

**Methods**

| | |
|---|---|
| *close* | Closes the underlying cluster. |
| *is_closed* | Property that determines whether the Engine's Client's resources are shutdown. |
| *send_data_to_cluster* | Send data to the cluster. |
| *setup_job_log* | Set up logger for job. |
| *submit_evaluation_job* | Send evaluation job to cluster. |
| *submit_scoring_job* | Send scoring job to cluster. |
| *submit_training_job* | Send training job to cluster. |

**close**(*self*)

Closes the underlying cluster.

**property is_closed**(*self*)

Property that determines whether the Engine's Client's resources are shutdown.

**send_data_to_cluster**(*self*, *X*, *y*)

Send data to the cluster.

The implementation uses caching so the data is only sent once. This follows dask best practices.

**Parameters**

- **X** (*pd.DataFrame*) – Input data for modeling.
- **y** (*pd.Series*) – Target data for modeling.

**Returns** The modeling data.

**Return type** dask.Future

**static setup_job_log**()

Set up logger for job.

**submit_evaluation_job**(*self*, *automl_config*, *pipeline*, *X*, *y*, *X_holdout=None*, *y_holdout=None*)

Send evaluation job to cluster.

**Parameters**

- **automl_config** – Structure containing data passed from AutoMLSearch instance.
- **pipeline** (*pipeline.PipelineBase*) – Pipeline to evaluate.
- **X** (*pd.DataFrame*) – Input data for modeling.
- **y** (*pd.Series*) – Target data for modeling.
- **X_holdout** (*pd.Series*) – Holdout input data for holdout scoring.
- **y_holdout** (*pd.Series*) – Holdout target data for holdout scoring.

**Returns**

**An object wrapping a reference to a future-like computation** occurring in the dask cluster.

**Return type** *DaskComputation*

**submit_scoring_job**(*self*, *automl_config*, *pipeline*, *X*, *y*, *objectives*, *X_train=None*, *y_train=None*)

Send scoring job to cluster.

**Parameters**

- **automl_config** – Structure containing data passed from AutoMLSearch instance.
- **pipeline** (*pipeline.PipelineBase*) – Pipeline to train.
- **X** (*pd.DataFrame*) – Input data for modeling.
- **y** (*pd.Series*) – Target data for modeling.
- **X_train** (*pd.DataFrame*) – Training features. Used for feature engineering in time series.
- **y_train** (*pd.Series*) – Training target. Used for feature engineering in time series.
- **objectives** (*list[ObjectiveBase]*) – List of objectives to score on.

**Returns**

**An object wrapping a reference to a future-like computation** occurring in the dask cluster.

**Return type** *DaskComputation*

**submit_training_job**(*self*, *automl_config*, *pipeline*, *X*, *y*)

Send training job to cluster.

**Parameters**

- **automl_config** – Structure containing data passed from AutoMLSearch instance.
- **pipeline** (*pipeline.PipelineBase*) – Pipeline to train.
- **X** (*pd.DataFrame*) – Input data for modeling.
- **y** (*pd.Series*) – Target data for modeling.

**Returns**

**An object wrapping a reference to a future-like computation** occurring in the dask cluster.

**Return type** *DaskComputation*

## engine_base

Base class for EvalML engines.

### Module Contents

### Classes Summary

| | |
|---|---|
| *EngineBase* | Base class for EvalML engines. |
| *EngineComputation* | Wrapper around the result of a (possibly asynchronous) engine computation. |
| *JobLogger* | Mimic the behavior of a python logging.Logger but stores all messages rather than actually logging them. |

### Functions

| | |
|---|---|
| *evaluate_pipeline* | Function submitted to the submit_evaluation_job engine method. |
| *score_pipeline* | Wrap around pipeline.score method to make it easy to score pipelines with dask. |
| *train_and_score_pipeline* | Given a pipeline, config and data, train and score the pipeline and return the CV or TV scores. |
| *train_pipeline* | Train a pipeline and tune the threshold if necessary. |

### Contents

**class** evalml.automl.engine.engine_base.**EngineBase**

Base class for EvalML engines.

**Methods**

| | |
|---|---|
| *setup_job_log* | Set up logger for job. |
| *submit_evaluation_job* | Submit job for pipeline evaluation during AutoMLSearch. |
| *submit_scoring_job* | Submit job for pipeline scoring. |
| *submit_training_job* | Submit job for pipeline training. |

**static setup_job_log**()

Set up logger for job.

**abstract submit_evaluation_job**(*self*, *automl_config*, *pipeline*, *X*, *y*, *X_holdout=None*, *y_holdout=None*)

Submit job for pipeline evaluation during AutoMLSearch.

**abstract submit_scoring_job**(*self*, *automl_config*, *pipeline*, *X*, *y*, *objectives*, *X_train=None*, *y_train=None*)

Submit job for pipeline scoring.

**abstract submit_training_job**(*self*, *automl_config*, *pipeline*, *X*, *y*, *X_holdout=None*, *y_holdout=None*)

Submit job for pipeline training.

**class** evalml.automl.engine.engine_base.**EngineComputation**

Wrapper around the result of a (possibly asynchronous) engine computation.

**Methods**

| | |
|---|---|
| *cancel* | Cancel the computation. |
| *done* | Whether the computation is done. |
| *get_result* | Gets the computation result. Will block until the computation is finished. |

**abstract cancel**(*self*)

Cancel the computation.

**abstract done**(*self*)

Whether the computation is done.

**abstract get_result**(*self*)

Gets the computation result. Will block until the computation is finished.

Raises Exception: If computation fails. Returns traceback.

evalml.automl.engine.engine_base.**evaluate_pipeline**(*pipeline*, *automl_config*, *X*, *y*, *logger*, *X_holdout=None*, *y_holdout=None*)

Function submitted to the submit_evaluation_job engine method.

**Parameters**

- **pipeline** (*PipelineBase*) – The pipeline to score.

- **automl_config** (*AutoMLConfig*) – The AutoMLSearch object, used to access config and the error callback.
- **X** (*pd.DataFrame*) – Training features.
- **y** (*pd.Series*) – Training target.
- **logger** – Logger object to write to.
- **X_holdout** (*pd.DataFrame*) – Holdout set features.
- **y_holdout** (*pd.DataFrame*) – Holdout set target.

**Returns**

**First - A dict containing cv_score_mean, cv_scores, training_time and a cv_data structure with details.** Second - The pipeline class we trained and scored. Third - the job logger instance with all the recorded messages.

**Return type** tuple of three items

**class** evalml.automl.engine.engine_base.**JobLogger**

Mimic the behavior of a python logging.Logger but stores all messages rather than actually logging them.

This is used during engine jobs so that log messages are recorded after the job completes. This is desired so that all of the messages for a single job are grouped together in the log.

**Methods**

| | |
|---|---|
| *debug* | Store message at the debug level. |
| *error* | Store message at the error level. |
| *info* | Store message at the info level. |
| *warning* | Store message at the warning level. |
| *write_to_logger* | Write all the messages to the logger, first in, first out (FIFO) order. |

**debug**(*self*, *msg*)

Store message at the debug level.

**error**(*self*, *msg*)

Store message at the error level.

**info**(*self*, *msg*)

Store message at the info level.

**warning**(*self*, *msg*)

Store message at the warning level.

**write_to_logger**(*self*, *logger*)

Write all the messages to the logger, first in, first out (FIFO) order.

evalml.automl.engine.engine_base.**score_pipeline**(*pipeline*, *X*, *y*, *objectives*, *X_train=None*, *y_train=None*, *X_schema=None*, *y_schema=None*)

Wrap around pipeline.score method to make it easy to score pipelines with dask.

**Parameters**

- **pipeline** (*PipelineBase*) – The pipeline to score.
- **X** (*pd.DataFrame*) – Features to score on.
- **y** (*pd.Series*) – Target used to calculate scores.

- **objectives** (*list[ObjectiveBase]*) – List of objectives to score on.
- **X_train** (*pd.DataFrame*) – Training features. Used for feature engineering in time series.
- **y_train** (*pd.Series*) – Training target. Used for feature engineering in time series.
- **X_schema** (*ww.TableSchema*) – Schema for features. Defaults to None.
- **y_schema** (*ww.ColumnSchema*) – Schema for columns. Defaults to None.

**Returns** Dictionary object containing pipeline scores.

**Return type** dict

evalml.automl.engine.engine_base.**train_and_score_pipeline**(*pipeline*, *automl_config*, *full_X_train*, *full_y_train*, *logger*, *X_holdout=None*, *y_holdout=None*)

Given a pipeline, config and data, train and score the pipeline and return the CV or TV scores.

**Parameters**

- **pipeline** (*PipelineBase*) – The pipeline to score.
- **automl_config** (*AutoMLSearch*) – The AutoMLSearch object, used to access config and the error callback.
- **full_X_train** (*pd.DataFrame*) – Training features.
- **full_y_train** (*pd.Series*) – Training target.
- **logger** – Logger object to write to.
- **X_holdout** (*pd.DataFrame*) – Holdout set features.
- **y_holdout** (*pd.DataFrame*) – Holdout set target.

**Raises** **Exception** – If there are missing target values in the training set after data split.

**Returns**

**First - A dict containing cv_score_mean, cv_scores, training_time and a cv_data structure with details.** Second - The pipeline class we trained and scored. Third - the job logger instance with all the recorded messages.

**Return type** tuple of three items

evalml.automl.engine.engine_base.**train_pipeline**(*pipeline*, *X*, *y*, *automl_config*, *schema=True*, *get_hashes=False*)

Train a pipeline and tune the threshold if necessary.

**Parameters**

- **pipeline** (*PipelineBase*) – Pipeline to train.
- **X** (*pd.DataFrame*) – Features to train on.
- **y** (*pd.Series*) – Target to train on.
- **automl_config** (*AutoMLSearch*) – The AutoMLSearch object, used to access config and the error callback.
- **schema** (*bool*) – Whether to use the schemas for X and y. Defaults to True.
- **get_hashes** (*bool*) – Whether to return the hashes of the data used to train (and potentially threshold). Defaults to False

**Returns** A trained pipeline instance. hash (optional): The hash of the input data indices, only returned when get_hashes is True.

**Return type** pipeline (PipelineBase)

### sequential_engine

A Future-like api for jobs created by the SequentialEngine, an Engine that sequentially computes the submitted jobs.

### Module Contents

### Classes Summary

| | |
|---|---|
| *SequentialComputation* | A Future-like api for jobs created by the SequentialEngine, an Engine that sequentially computes the submitted jobs. |
| *SequentialEngine* | The default engine for the AutoML search. |

### Contents

**class** evalml.automl.engine.sequential_engine.**SequentialComputation**(*work*, ***kwargs*)

A Future-like api for jobs created by the SequentialEngine, an Engine that sequentially computes the submitted jobs.

In order to separate the engine from the AutoMLSearch loop, we need the sequential computations to behave the same way as concurrent computations from AutoMLSearch's point-of-view. One way to do this is by delaying the computation in the sequential engine until get_result is called. Since AutoMLSearch will call get_result only when the computation is "done", by always returning True in done() we make sure that get_result is called in the order that the jobs are submitted. So the computations happen sequentially!

**Parameters** **work** (*callable*) – Computation that should be done by the engine.

**Methods**

| | |
|---|---|
| *cancel* | Cancel the current computation. |
| *done* | Whether the computation is done. |
| *get_result* | Gets the computation result. Will block until the computation is finished. |

**cancel**(*self*)

Cancel the current computation.

**done**(*self*)

Whether the computation is done.

**Returns** Always returns True.

**Return type** bool

**get_result**(*self*)

Gets the computation result. Will block until the computation is finished.

**Raises** **Exception** – If computation fails. Returns traceback.

**Returns** Computation results.

**class** evalml.automl.engine.sequential_engine.**SequentialEngine**

The default engine for the AutoML search.

Trains and scores pipelines locally and sequentially.

**Methods**

| | |
|---|---|
| *close* | No-op. |
| *setup_job_log* | Set up logger for job. |
| *submit_evaluation_job* | Submit a job to evaluate a pipeline. |
| *submit_scoring_job* | Submit a job to score a pipeline. |
| *submit_training_job* | Submit a job to train a pipeline. |

**close**(*self*)

No-op.

**static setup_job_log**()

Set up logger for job.

**submit_evaluation_job**(*self*, *automl_config*, *pipeline*, *X*, *y*, *X_holdout=None*, *y_holdout=None*)

Submit a job to evaluate a pipeline.

**Parameters**

- **automl_config** – Structure containing data passed from AutoMLSearch instance.
- **pipeline** (*pipeline.PipelineBase*) – Pipeline to evaluate.
- **X** (*pd.DataFrame*) – Input data for modeling.
- **y** (*pd.Series*) – Target data for modeling.
- **X_holdout** (*pd.Series*) – Holdout input data for holdout scoring.
- **y_holdout** (*pd.Series*) – Holdout target data for holdout scoring.

**Returns** Computation result.

**Return type** *SequentialComputation*

**submit_scoring_job**(*self*, *automl_config*, *pipeline*, *X*, *y*, *objectives*, *X_train=None*, *y_train=None*)

Submit a job to score a pipeline.

**Parameters**

- **automl_config** – Structure containing data passed from AutoMLSearch instance.
- **pipeline** (*pipeline.PipelineBase*) – Pipeline to train.
- **X** (*pd.DataFrame*) – Input data for modeling.
- **y** (*pd.Series*) – Target data for modeling.
- **X_train** (*pd.DataFrame*) – Training features. Used for feature engineering in time series.
- **y_train** (*pd.Series*) – Training target. Used for feature engineering in time series.
- **objectives** (*list[ObjectiveBase]*) – List of objectives to score on.

**Returns** Computation result.

**Return type** *SequentialComputation*

**submit_training_job**(*self*, *automl_config*, *pipeline*, *X*, *y*)

Submit a job to train a pipeline.

**Parameters**

- **automl_config** – Structure containing data passed from AutoMLSearch instance.
- **pipeline** (*pipeline.PipelineBase*) – Pipeline to evaluate.
- **X** (*pd.DataFrame*) – Input data for modeling.
- **y** (*pd.Series*) – Target data for modeling.

**Returns** Computation result.

**Return type** *SequentialComputation*

## Package Contents

## Classes Summary

| | |
|---|---|
| *CFEngine* | The concurrent.futures (CF) engine. |
| *DaskEngine* | The dask engine. |
| *EngineBase* | Base class for EvalML engines. |
| *EngineComputation* | Wrapper around the result of a (possibly asynchronous) engine computation. |
| *SequentialEngine* | The default engine for the AutoML search. |

## Functions

| | |
|---|---|
| *evaluate_pipeline* | Function submitted to the submit_evaluation_job engine method. |
| *train_and_score_pipeline* | Given a pipeline, config and data, train and score the pipeline and return the CV or TV scores. |
| *train_pipeline* | Train a pipeline and tune the threshold if necessary. |

## Contents

**class** evalml.automl.engine.**CFEngine**(*client=None*)

The concurrent.futures (CF) engine.

**Parameters** **client** (*None or CFClient*) – If None, creates a threaded pool for processing. Defaults to None.

**Methods**

| | |
|---|---|
| *close* | Function to properly shutdown the Engine's Client's resources. |
| *is_closed* | Property that determines whether the Engine's Client's resources are shutdown. |
| *setup_job_log* | Set up logger for job. |
| *submit_evaluation_job* | Send evaluation job to cluster. |
| *submit_scoring_job* | Send scoring job to cluster. |
| *submit_training_job* | Send training job to cluster. |

**close**(*self*)

Function to properly shutdown the Engine's Client's resources.

**property is_closed**(*self*)

Property that determines whether the Engine's Client's resources are shutdown.

**static setup_job_log**()

Set up logger for job.

**submit_evaluation_job**(*self*, *automl_config*, *pipeline*, *X*, *y*, *X_holdout=None*, *y_holdout=None*)

Send evaluation job to cluster.

**Parameters**

- **automl_config** – Structure containing data passed from AutoMLSearch instance.
- **pipeline** (*pipeline.PipelineBase*) – Pipeline to evaluate.
- **X** (*pd.DataFrame*) – Input data for modeling.
- **y** (*pd.Series*) – Target data for modeling.
- **X_holdout** (*pd.Series*) – Holdout input data for holdout scoring.
- **y_holdout** (*pd.Series*) – Holdout target data for holdout scoring.

**Returns**

**An object wrapping a reference to a future-like computation** occurring in the resource pool

**Return type** CFComputation

**submit_scoring_job**(*self*, *automl_config*, *pipeline*, *X*, *y*, *objectives*, *X_train=None*, *y_train=None*)

Send scoring job to cluster.

**Parameters**

- **automl_config** – Structure containing data passed from AutoMLSearch instance.
- **pipeline** (*pipeline.PipelineBase*) – Pipeline to train.
- **X** (*pd.DataFrame*) – Input data for modeling.
- **y** (*pd.Series*) – Target data for modeling.
- **X_train** (*pd.DataFrame*) – Training features. Used for feature engineering in time series.
- **y_train** (*pd.Series*) – Training target. Used for feature engineering in time series.
- **objectives** (*list[ObjectiveBase]*) – Objectives to score on.

**Returns**

**An object wrapping a reference to a future-like computation** occurring in the resource pool.

**Return type** CFComputation

**submit_training_job**(*self*, *automl_config*, *pipeline*, *X*, *y*)

Send training job to cluster.

**Parameters**

- **automl_config** – Structure containing data passed from AutoMLSearch instance.
- **pipeline** (*pipeline.PipelineBase*) – Pipeline to train.
- **X** (*pd.DataFrame*) – Input data for modeling.
- **y** (*pd.Series*) – Target data for modeling.

**Returns**

**An object wrapping a reference to a future-like computation** occurring in the resource pool

**Return type** CFComputation

**class** evalml.automl.engine.**DaskEngine**(*cluster=None*)

The dask engine.

**Parameters** **cluster** (*None or dd.Client*) – If None, creates a local, threaded Dask client for processing. Defaults to None.

**Methods**

| | |
|---|---|
| *close* | Closes the underlying cluster. |
| *is_closed* | Property that determines whether the Engine's Client's resources are shutdown. |
| *send_data_to_cluster* | Send data to the cluster. |
| *setup_job_log* | Set up logger for job. |
| *submit_evaluation_job* | Send evaluation job to cluster. |
| *submit_scoring_job* | Send scoring job to cluster. |
| *submit_training_job* | Send training job to cluster. |

**close**(*self*)

Closes the underlying cluster.

**property is_closed**(*self*)

Property that determines whether the Engine's Client's resources are shutdown.

**send_data_to_cluster**(*self*, *X*, *y*)

Send data to the cluster.

The implementation uses caching so the data is only sent once. This follows dask best practices.

**Parameters**

- **X** (*pd.DataFrame*) – Input data for modeling.
- **y** (*pd.Series*) – Target data for modeling.

**Returns** The modeling data.

**Return type** dask.Future

**static setup_job_log()**

Set up logger for job.

**submit_evaluation_job**(*self*, *automl_config*, *pipeline*, *X*, *y*, *X_holdout=None*, *y_holdout=None*)

Send evaluation job to cluster.

**Parameters**

- **automl_config** – Structure containing data passed from AutoMLSearch instance.
- **pipeline** (*pipeline.PipelineBase*) – Pipeline to evaluate.
- **X** (*pd.DataFrame*) – Input data for modeling.
- **y** (*pd.Series*) – Target data for modeling.
- **X_holdout** (*pd.Series*) – Holdout input data for holdout scoring.
- **y_holdout** (*pd.Series*) – Holdout target data for holdout scoring.

**Returns**

**An object wrapping a reference to a future-like computation** occurring in the dask cluster.

**Return type** DaskComputation

**submit_scoring_job**(*self*, *automl_config*, *pipeline*, *X*, *y*, *objectives*, *X_train=None*, *y_train=None*)

Send scoring job to cluster.

**Parameters**

- **automl_config** – Structure containing data passed from AutoMLSearch instance.
- **pipeline** (*pipeline.PipelineBase*) – Pipeline to train.
- **X** (*pd.DataFrame*) – Input data for modeling.
- **y** (*pd.Series*) – Target data for modeling.
- **X_train** (*pd.DataFrame*) – Training features. Used for feature engineering in time series.
- **y_train** (*pd.Series*) – Training target. Used for feature engineering in time series.
- **objectives** (*list[ObjectiveBase]*) – List of objectives to score on.

**Returns**

**An object wrapping a reference to a future-like computation** occurring in the dask cluster.

**Return type** DaskComputation

**submit_training_job**(*self*, *automl_config*, *pipeline*, *X*, *y*)

Send training job to cluster.

**Parameters**

- **automl_config** – Structure containing data passed from AutoMLSearch instance.
- **pipeline** (*pipeline.PipelineBase*) – Pipeline to train.
- **X** (*pd.DataFrame*) – Input data for modeling.
- **y** (*pd.Series*) – Target data for modeling.

**Returns**

**An object wrapping a reference to a future-like computation** occurring in the dask cluster.

**Return type** DaskComputation

**class** evalml.automl.engine.**EngineBase**

Base class for EvalML engines.

**Methods**

| | |
|---|---|
| *setup_job_log* | Set up logger for job. |
| *submit_evaluation_job* | Submit job for pipeline evaluation during AutoMLSearch. |
| *submit_scoring_job* | Submit job for pipeline scoring. |
| *submit_training_job* | Submit job for pipeline training. |

**static setup_job_log**()

Set up logger for job.

**abstract submit_evaluation_job**(*self*, *automl_config*, *pipeline*, *X*, *y*, *X_holdout=None*, *y_holdout=None*)

Submit job for pipeline evaluation during AutoMLSearch.

**abstract submit_scoring_job**(*self*, *automl_config*, *pipeline*, *X*, *y*, *objectives*, *X_train=None*, *y_train=None*)

Submit job for pipeline scoring.

**abstract submit_training_job**(*self*, *automl_config*, *pipeline*, *X*, *y*, *X_holdout=None*, *y_holdout=None*)

Submit job for pipeline training.

**class** evalml.automl.engine.**EngineComputation**

Wrapper around the result of a (possibly asynchronous) engine computation.

**Methods**

| | |
|---|---|
| *cancel* | Cancel the computation. |
| *done* | Whether the computation is done. |
| *get_result* | Gets the computation result. Will block until the computation is finished. |

**abstract cancel**(*self*)

Cancel the computation.

**abstract done**(*self*)

Whether the computation is done.

**abstract get_result**(*self*)

Gets the computation result. Will block until the computation is finished.

Raises Exception: If computation fails. Returns traceback.

evalml.automl.engine.**evaluate_pipeline**(*pipeline*, *automl_config*, *X*, *y*, *logger*, *X_holdout=None*, *y_holdout=None*)

Function submitted to the submit_evaluation_job engine method.

**Parameters**

- **pipeline** (*PipelineBase*) – The pipeline to score.
- **automl_config** (*AutoMLConfig*) – The AutoMLSearch object, used to access config and the error callback.
- **X** (*pd.DataFrame*) – Training features.
- **y** (*pd.Series*) – Training target.
- **logger** – Logger object to write to.
- **X_holdout** (*pd.DataFrame*) – Holdout set features.
- **y_holdout** (*pd.DataFrame*) – Holdout set target.

**Returns**

**First - A dict containing cv_score_mean, cv_scores, training_time and a cv_data structure with details.** Second - The pipeline class we trained and scored. Third - the job logger instance with all the recorded messages.

**Return type** tuple of three items

**class** evalml.automl.engine.**SequentialEngine**

The default engine for the AutoML search.

Trains and scores pipelines locally and sequentially.

**Methods**

| | |
|---|---|
| *close* | No-op. |
| *setup_job_log* | Set up logger for job. |
| *submit_evaluation_job* | Submit a job to evaluate a pipeline. |
| *submit_scoring_job* | Submit a job to score a pipeline. |
| *submit_training_job* | Submit a job to train a pipeline. |

**close**(*self*)

No-op.

**static setup_job_log**()

Set up logger for job.

**submit_evaluation_job**(*self*, *automl_config*, *pipeline*, *X*, *y*, *X_holdout=None*, *y_holdout=None*)

Submit a job to evaluate a pipeline.

**Parameters**

- **automl_config** – Structure containing data passed from AutoMLSearch instance.
- **pipeline** (*pipeline.PipelineBase*) – Pipeline to evaluate.
- **X** (*pd.DataFrame*) – Input data for modeling.
- **y** (*pd.Series*) – Target data for modeling.
- **X_holdout** (*pd.Series*) – Holdout input data for holdout scoring.
- **y_holdout** (*pd.Series*) – Holdout target data for holdout scoring.

**Returns** Computation result.

**Return type** SequentialComputation

**submit_scoring_job**(*self*, *automl_config*, *pipeline*, *X*, *y*, *objectives*, *X_train=None*, *y_train=None*)

Submit a job to score a pipeline.

**Parameters**

- **automl_config** – Structure containing data passed from AutoMLSearch instance.
- **pipeline** (*pipeline.PipelineBase*) – Pipeline to train.
- **X** (*pd.DataFrame*) – Input data for modeling.
- **y** (*pd.Series*) – Target data for modeling.
- **X_train** (*pd.DataFrame*) – Training features. Used for feature engineering in time series.
- **y_train** (*pd.Series*) – Training target. Used for feature engineering in time series.
- **objectives** (*list[ObjectiveBase]*) – List of objectives to score on.

**Returns** Computation result.

**Return type** SequentialComputation

**submit_training_job**(*self*, *automl_config*, *pipeline*, *X*, *y*)

Submit a job to train a pipeline.

**Parameters**

- **automl_config** – Structure containing data passed from AutoMLSearch instance.
- **pipeline** (*pipeline.PipelineBase*) – Pipeline to evaluate.
- **X** (*pd.DataFrame*) – Input data for modeling.
- **y** (*pd.Series*) – Target data for modeling.

**Returns** Computation result.

**Return type** SequentialComputation

evalml.automl.engine.**train_and_score_pipeline**(*pipeline*, *automl_config*, *full_X_train*, *full_y_train*, *logger*, *X_holdout=None*, *y_holdout=None*)

Given a pipeline, config and data, train and score the pipeline and return the CV or TV scores.

**Parameters**

- **pipeline** (*PipelineBase*) – The pipeline to score.
- **automl_config** (*AutoMLSearch*) – The AutoMLSearch object, used to access config and the error callback.
- **full_X_train** (*pd.DataFrame*) – Training features.
- **full_y_train** (*pd.Series*) – Training target.
- **logger** – Logger object to write to.
- **X_holdout** (*pd.DataFrame*) – Holdout set features.
- **y_holdout** (*pd.DataFrame*) – Holdout set target.

**Raises** **Exception** – If there are missing target values in the training set after data split.

**Returns**

**First - A dict containing cv_score_mean, cv_scores, training_time and a cv_data structure with details.** Second - The pipeline class we trained and scored. Third - the job logger instance with all the recorded messages.

**Return type** tuple of three items

`evalml.automl.engine.train_pipeline(pipeline, X, y, automl_config, schema=True, get_hashes=False)`

Train a pipeline and tune the threshold if necessary.

**Parameters**

- **pipeline** (*PipelineBase*) – Pipeline to train.
- **X** (*pd.DataFrame*) – Features to train on.
- **y** (*pd.Series*) – Target to train on.
- **automl_config** (*AutoMLSearch*) – The AutoMLSearch object, used to access config and the error callback.
- **schema** (*bool*) – Whether to use the schemas for X and y. Defaults to True.
- **get_hashes** (*bool*) – Whether to return the hashes of the data used to train (and potentially threshold). Defaults to False

**Returns** A trained pipeline instance. hash (optional): The hash of the input data indices, only returned when get_hashes is True.

**Return type** pipeline (PipelineBase)

## Submodules

### automl_search

EvalML's core AutoML object.

## Module Contents

## Classes Summary

| | |
|---|---|
| *AutoMLSearch* | Automated Pipeline search. |

## Functions

| | |
|---|---|
| *build_engine_from_str* | Function that converts a convenience string for an parallel engine type and returns an instance of that engine. |
| *search* | Given data and configuration, run an automl search. |
| *search_iterative* | Given data and configuration, run an automl search. |

## Contents

**class** evalml.automl.automl_search.**AutoMLSearch**(*X_train=None*, *y_train=None*, *X_holdout=None*, *y_holdout=None*, *problem_type=None*, *objective='auto'*, *max_iterations=None*, *max_time=None*, *patience=None*, *tolerance=None*, *data_splitter=None*, *allowed_component_graphs=None*, *allowed_model_families=None*, *features=None*, *start_iteration_callback=None*, *add_result_callback=None*, *error_callback=None*, *additional_objectives=None*, *alternate_thresholding_objective='F1'*, *random_seed=0*, *n_jobs=- 1*, *tuner_class=None*, *optimize_thresholds=True*, *ensembling=False*, *max_batches=None*, *problem_configuration=None*, *train_best_pipeline=True*, *search_parameters=None*, *sampler_method='auto'*, *sampler_balanced_ratio=0.25*, *allow_long_running_models=False*, *_pipelines_per_batch=5*, *automl_algorithm='default'*, *engine='sequential'*, *verbose=False*, *timing=False*, *exclude_featurizers=None*, *holdout_set_size=0*)

Automated Pipeline search.

**Parameters**

- **X_train** (*pd.DataFrame*) – The input training data of shape [n_samples, n_features]. Required.
- **y_train** (*pd.Series*) – The target training data of length [n_samples]. Required for supervised learning tasks.
- **X_holdout** (*pd.DataFrame*) – The input holdout data of shape [n_samples, n_features].
- **y_holdout** (*pd.Series*) – The target holdout data of length [n_samples].
- **problem_type** (*str or ProblemTypes*) – Type of supervised learning problem. See evalml.problem_types.ProblemType.all_problem_types for a full list.
- **objective** (*str, ObjectiveBase*) – The objective to optimize for. Used to propose and rank pipelines, but not for optimizing each pipeline during fit-time. When set to 'auto', chooses: - LogLossBinary for binary classification problems, - LogLossMulticlass for multiclass classification problems, and - R2 for regression problems.
- **max_iterations** (*int*) – Maximum number of iterations to search. If max_iterations and max_time is not set, then max_iterations will default to max_iterations of 5.
- **max_time** (*int, str*) – Maximum time to search for pipelines. This will not start a new pipeline search after the duration has elapsed. If it is an integer, then the time will be in seconds. For strings, time can be specified as seconds, minutes, or hours.
- **patience** (*int*) – Number of iterations without improvement to stop search early. Must be positive. If None, early stopping is disabled. Defaults to None.
- **tolerance** (*float*) – Minimum percentage difference to qualify as score improvement for early stopping. Only applicable if patience is not None. Defaults to None.

- **allowed_component_graphs** (*dict*) – A dictionary of lists or ComponentGraphs indicating the component graphs allowed in the search. The format should follow { "Name_0": [list_of_components], "Name_1": ComponentGraph(...) }

  The default of None indicates all pipeline component graphs for this problem type are allowed. Setting this field will cause allowed_model_families to be ignored.

  e.g. allowed_component_graphs = { "My_Graph": ["Imputer", "One Hot Encoder", "Random Forest Classifier"] }

- **allowed_model_families** (*list(str, ModelFamily)*) – The model families to search. The default of None searches over all model families. Run evalml.pipelines.components.utils.allowed_model_families("binary") to see options. Change *binary* to *multiclass* or *regression* depending on the problem type. Note that if allowed_pipelines is provided, this parameter will be ignored.

- **features** (*list*) – List of features to run DFS on AutoML pipelines. Defaults to None. Features will only be computed if the columns used by the feature exist in the search input and if the feature itself is not in search input. If features is an empty list, the DFS Transformer will not be included in pipelines.

- **data_splitter** (*sklearn.model_selection.BaseCrossValidator*) – Data splitting method to use. Defaults to StratifiedKFold.

- **tuner_class** – The tuner class to use. Defaults to SKOptTuner.

- **optimize_thresholds** (*bool*) – Whether or not to optimize the binary pipeline threshold. Defaults to True.

- **start_iteration_callback** (*callable*) – Function called before each pipeline training iteration. Callback function takes three positional parameters: The pipeline instance and the AutoMLSearch object.

- **add_result_callback** (*callable*) – Function called after each pipeline training iteration. Callback function takes three positional parameters: A dictionary containing the training results for the new pipeline, an untrained_pipeline containing the parameters used during training, and the AutoMLSearch object.

- **error_callback** (*callable*) – Function called when *search()* errors and raises an Exception. Callback function takes three positional parameters: the Exception raised, the traceback, and the AutoMLSearch object. Must also accepts kwargs, so AutoMLSearch is able to pass along other appropriate parameters by default. Defaults to None, which will call *log_error_callback*.

- **additional_objectives** (*list*) – Custom set of objectives to score on. Will override default objectives for problem type if not empty.

- **alternate_thresholding_objective** (*str*) – The objective to use for thresholding binary classification pipelines if the main objective provided isn't tuneable. Defaults to F1.

- **random_seed** (*int*) – Seed for the random number generator. Defaults to 0.

- **n_jobs** (*int or None*) – Non-negative integer describing level of parallelism used for pipelines. None and 1 are equivalent. If set to -1, all CPUs are used. For n_jobs below -1, (n_cpus + 1 + n_jobs) are used.

- **ensembling** (*boolean*) – If True, runs ensembling in a separate batch after every allowed pipeline class has been iterated over. If the number of unique pipelines to search over per batch is one, ensembling will not run. Defaults to False.

- **max_batches** (*int*) – The maximum number of batches of pipelines to search. Parameters max_time, and max_iterations have precedence over stopping the search.

- **problem_configuration** (*dict, None*) – Additional parameters needed to configure the search. For example, in time series problems, values should be passed in for the time_index, gap, forecast_horizon, and max_delay variables.
- **train_best_pipeline** (*boolean*) – Whether or not to train the best pipeline before returning it. Defaults to True.
- **search_parameters** (*dict*) – A dict of the hyperparameter ranges or pipeline parameters used to iterate over during search. Keys should consist of the component names and values should specify a singular value/list for pipeline parameters, or skopt.Space for hyperparameter ranges. In the example below, the Imputer parameters would be passed to the hyperparameter ranges, and the Label Encoder parameters would be used as the component parameter.

  **e.g. search_parameters = { 'Imputer'** [{ 'numeric_impute_strategy': Categorical(['most_frequent', 'median']) },] 'Label Encoder': {'positive_label': True} }
- **sampler_method** (*str*) – The data sampling component to use in the pipelines if the problem type is classification and the target balance is smaller than the sampler_balanced_ratio. Either 'auto', which will use our preferred sampler for the data, 'Undersampler', 'Oversampler', or None. Defaults to 'auto'.
- **sampler_balanced_ratio** (*float*) – The minority:majority class ratio that we consider balanced, so a 1:4 ratio would be equal to 0.25. If the class balance is larger than this provided value, then we will not add a sampler since the data is then considered balanced. Overrides the *sampler_ratio* of the samplers. Defaults to 0.25.
- **allow_long_running_models** (*bool*) – Whether or not to allow longer-running models for large multiclass problems. If False and no pipelines, component graphs, or model families are provided, AutoMLSearch will not use Elastic Net or XGBoost when there are more than 75 multiclass targets and will not use CatBoost when there are more than 150 multiclass targets. Defaults to False.
- **_ensembling_split_size** (*float*) – The amount of the training data we'll set aside for training ensemble metalearners. Only used when ensembling is True. Must be between 0 and 1, exclusive. Defaults to 0.2
- **_pipelines_per_batch** (*int*) – The number of pipelines to train for every batch after the first one. The first batch will train a baseline pipline + one of each pipeline family allowed in the search.
- **automl_algorithm** (*str*) – The automl algorithm to use. Currently the two choices are 'iterative' and 'default'. Defaults to *default*.
- **engine** (*EngineBase or str*) – The engine instance used to evaluate pipelines. Dask or concurrent.futures engines can also be chosen by providing a string from the list ["sequential", "cf_threaded", "cf_process", "dask_threaded", "dask_process"]. If a parallel engine is selected this way, the maximum amount of parallelism, as determined by the engine, will be used. Defaults to "sequential".
- **verbose** (*boolean*) – Whether or not to display semi-real-time updates to stdout while search is running. Defaults to False.
- **timing** (*boolean*) – Whether or not to write pipeline search times to the logger. Defaults to False.
- **exclude_featurizers** (*list[str]*) – A list of featurizer components to exclude from the pipelines built by search. Valid options are "DatetimeFeaturizer", "EmailFeaturizer", "URLFeaturizer", "NaturalLanguageFeaturizer", "TimeSeriesFeaturizer"

- **holdout_set_size** (*float*) – The size of the holdout set that AutoML search will take for datasets larger than 500 rows. If set to 0, holdout set will not be taken regardless of number of rows. Must be between 0 and 1, exclusive. Defaults to 0.1.

**Methods**

| | |
|---|---|
| *add_to_rankings* | Fits and evaluates a given pipeline then adds the results to the automl rankings with the requirement that automl search has been run. |
| *best_pipeline* | Returns a trained instance of the best pipeline and parameters found during automl search. If *train_best_pipeline* is set to False, returns an untrained pipeline instance. |
| *close_engine* | Function to explicitly close the engine, client, parallel resources. |
| *describe_pipeline* | Describe a pipeline. |
| *full_rankings* | Returns a pandas.DataFrame with scoring results from all pipelines searched. |
| *get_ensembler_input_pipelines* | Returns a list of input pipeline IDs given an ensembler pipeline ID. |
| *get_pipeline* | Given the ID of a pipeline training result, returns an untrained instance of the specified pipeline initialized with the parameters used to train that pipeline during automl search. |
| *load* | Loads AutoML object at file path. |
| *plot* | Return an instance of the plot with the latest scores. |
| *rankings* | Returns a pandas.DataFrame with scoring results from the highest-scoring set of parameters used with each pipeline. |
| *results* | Class that allows access to a copy of the results from *automl_search*. |
| *save* | Saves AutoML object at file path. |
| *score_pipelines* | Score a list of pipelines on the given holdout data. |
| *search* | Find the best pipeline for the data set. |
| *train_pipelines* | Train a list of pipelines on the training data. |

**add_to_rankings**(*self*, *pipeline*)

Fits and evaluates a given pipeline then adds the results to the automl rankings with the requirement that automl search has been run.

**Parameters** **pipeline** (*PipelineBase*) – pipeline to train and evaluate.

**property best_pipeline**(*self*)

Returns a trained instance of the best pipeline and parameters found during automl search. If *train_best_pipeline* is set to False, returns an untrained pipeline instance.

**Returns** A trained instance of the best pipeline and parameters found during automl search. If *train_best_pipeline* is set to False, returns an untrained pipeline instance.

**Return type** PipelineBase

**Raises** **PipelineNotFoundError** – If this is called before .search() is called.

**close_engine**(*self*)

Function to explicitly close the engine, client, parallel resources.

**describe_pipeline**(*self*, *pipeline_id*, *return_dict=False*)

Describe a pipeline.

**Parameters**

- **pipeline_id** (*int*) – pipeline to describe
- **return_dict** (*bool*) – If True, return dictionary of information about pipeline. Defaults to False.

**Returns** Description of specified pipeline. Includes information such as type of pipeline components, problem, training time, cross validation, etc.

**Raises** **PipelineNotFoundError** – If pipeline_id is not a valid ID.

**property full_rankings**(*self*)

Returns a pandas.DataFrame with scoring results from all pipelines searched.

**get_ensembler_input_pipelines**(*self*, *ensemble_pipeline_id*)

Returns a list of input pipeline IDs given an ensembler pipeline ID.

**Parameters** **ensemble_pipeline_id** (*id*) – Ensemble pipeline ID to get input pipeline IDs from.

**Returns** A list of ensemble input pipeline IDs.

**Return type** list[int]

**Raises** **ValueError** – If *ensemble_pipeline_id* does not correspond to a valid ensemble pipeline ID.

**get_pipeline**(*self*, *pipeline_id*)

Given the ID of a pipeline training result, returns an untrained instance of the specified pipeline initialized with the parameters used to train that pipeline during automl search.

**Parameters** **pipeline_id** (*int*) – Pipeline to retrieve.

**Returns** Untrained pipeline instance associated with the provided ID.

**Return type** PipelineBase

**Raises** **PipelineNotFoundError** – if pipeline_id is not a valid ID.

**static load**(*file_path*, *pickle_type='cloudpickle'*)

Loads AutoML object at file path.

**Parameters**

- **file_path** (*str*) – Location to find file to load
- **pickle_type** (*{"pickle", "cloudpickle"}*) – The pickling library to use. Currently not used since the standard pickle library can handle cloudpickles.

**Returns** AutoSearchBase object

**property plot**(*self*)

Return an instance of the plot with the latest scores.

**property rankings**(*self*)

Returns a pandas.DataFrame with scoring results from the highest-scoring set of parameters used with each pipeline.

**property results**(*self*)

Class that allows access to a copy of the results from *automl_search*.

**Returns**

**Dictionary containing *pipeline_results*, a dict with results from each pipeline,** and *search_order*, a list describing the order the pipelines were searched.

**Return type** dict

**save**(*self*, *file_path*, *pickle_type='cloudpickle'*, *pickle_protocol=cloudpickle.DEFAULT_PROTOCOL*)

Saves AutoML object at file path.

**Parameters**

- **file_path** (*str*) – Location to save file.
- **pickle_type** (*{"pickle", "cloudpickle"}*) – The pickling library to use.
- **pickle_protocol** (*int*) – The pickle data stream format.

**Raises** **ValueError** – If pickle_type is not "pickle" or "cloudpickle".

**score_pipelines**(*self*, *pipelines*, *X_holdout*, *y_holdout*, *objectives*)

Score a list of pipelines on the given holdout data.

**Parameters**

- **pipelines** (*list[PipelineBase]*) – List of pipelines to train.
- **X_holdout** (*pd.DataFrame*) – Holdout features.
- **y_holdout** (*pd.Series*) – Holdout targets for scoring.
- **objectives** (*list[str], list[ObjectiveBase]*) – Objectives used for scoring.

**Returns** Dictionary keyed by pipeline name that maps to a dictionary of scores. Note that the any pipelines that error out during scoring will not be included in the dictionary but the exception and stacktrace will be displayed in the log.

**Return type** dict[str, Dict[str, float]]

**search**(*self*, *interactive_plot=True*)

Find the best pipeline for the data set.

**Parameters** **interactive_plot** (*boolean, True*) – Shows an iteration vs. score plot in Jupyter notebook. Disabled by default in non-Jupyter enviroments.

**Raises** **AutoMLSearchException** – If all pipelines in the current AutoML batch produced a score of np.nan on the primary objective.

**Returns** Dictionary keyed by batch number that maps to the timings for pipelines run in that batch, as well as the total time for each batch. Pipelines within a batch are labeled by pipeline name.

**Return type** Dict[int, Dict[str, Timestamp]]

**train_pipelines**(*self*, *pipelines*)

Train a list of pipelines on the training data.

This can be helpful for training pipelines once the search is complete.

**Parameters** **pipelines** (*list[PipelineBase]*) – List of pipelines to train.

**Returns** Dictionary keyed by pipeline name that maps to the fitted pipeline. Note that the any pipelines that error out during training will not be included in the dictionary but the exception and stacktrace will be displayed in the log.

**Return type** Dict[str, PipelineBase]

`evalml.automl.automl_search.build_engine_from_str(engine_str)`

Function that converts a convenience string for an parallel engine type and returns an instance of that engine.

**Parameters** **engine_str** (*str*) – String representing the requested engine.

**Returns** Instance of the requested engine.

**Return type** (EngineBase)

**Raises** **ValueError** – If engine_str is not a valid engine.

`evalml.automl.automl_search.search(X_train=None, y_train=None, problem_type=None, objective='auto', mode='fast', max_time=None, patience=None, tolerance=None, problem_configuration=None, n_splits=3, verbose=False, timing=False)`

Given data and configuration, run an automl search.

This method will run EvalML's default suite of data checks. If the data checks produce errors, the data check results will be returned before running the automl search. In that case we recommend you alter your data to address these errors and try again. This method is provided for convenience. If you'd like more control over when each of these steps is run, consider making calls directly to the various pieces like the data checks and AutoMLSearch, instead of using this method.

**Parameters**

- **X_train** (*pd.DataFrame*) – The input training data of shape [n_samples, n_features]. Required.
- **y_train** (*pd.Series*) – The target training data of length [n_samples]. Required for supervised learning tasks.
- **problem_type** (*str or ProblemTypes*) – Type of supervised learning problem. See evalml.problem_types.ProblemType.all_problem_types for a full list.
- **objective** (*str, ObjectiveBase*) – The objective to optimize for. Used to propose and rank pipelines, but not for optimizing each pipeline during fit-time. When set to 'auto', chooses: - LogLossBinary for binary classification problems, - LogLossMulticlass for multiclass classification problems, and - R2 for regression problems.
- **mode** (*str*) – mode for DefaultAlgorithm. There are two modes: fast and long, where fast is a subset of long. Please look at DefaultAlgorithm for more details.
- **max_time** (*int, str*) – Maximum time to search for pipelines. This will not start a new pipeline search after the duration has elapsed. If it is an integer, then the time will be in seconds. For strings, time can be specified as seconds, minutes, or hours.
- **patience** (*int*) – Number of iterations without improvement to stop search early. Must be positive. If None, early stopping is disabled. Defaults to None.
- **tolerance** (*float*) – Minimum percentage difference to qualify as score improvement for early stopping. Only applicable if patience is not None. Defaults to None.
- **problem_configuration** (*dict*) – Additional parameters needed to configure the search. For example, in time series problems, values should be passed in for the time_index, gap, forecast_horizon, and max_delay variables.
- **n_splits** (*int*) – Number of splits to use with the default data splitter.

- **verbose** (*boolean*) – Whether or not to display semi-real-time updates to stdout while search is running. Defaults to False.
- **timing** (*boolean*) – Whether or not to write pipeline search times to the logger. Defaults to False.

**Returns** The automl search object containing pipelines and rankings, and the results from running the data checks. If the data check results contain errors, automl search will not be run and an automl search object will not be returned.

**Return type** (*AutoMLSearch*, dict)

**Raises** **ValueError** – If search configuration is not valid.

evalml.automl.automl_search.**search_iterative**(*X_train=None*, *y_train=None*, *problem_type=None*, *objective='auto'*, *problem_configuration=None*, *n_splits=3*, *timing=False*, ***kwargs*)

Given data and configuration, run an automl search.

This method will run EvalML's default suite of data checks. If the data checks produce errors, the data check results will be returned before running the automl search. In that case we recommend you alter your data to address these errors and try again. This method is provided for convenience. If you'd like more control over when each of these steps is run, consider making calls directly to the various pieces like the data checks and AutoMLSearch, instead of using this method.

**Parameters**

- **X_train** (*pd.DataFrame*) – The input training data of shape [n_samples, n_features]. Required.
- **y_train** (*pd.Series*) – The target training data of length [n_samples]. Required for supervised learning tasks.
- **problem_type** (*str or ProblemTypes*) – Type of supervised learning problem. See evalml.problem_types.ProblemType.all_problem_types for a full list.
- **objective** (*str, ObjectiveBase*) – The objective to optimize for. Used to propose and rank pipelines, but not for optimizing each pipeline during fit-time. When set to 'auto', chooses: - LogLossBinary for binary classification problems, - LogLossMulticlass for multiclass classification problems, and - R2 for regression problems.
- **problem_configuration** (*dict*) – Additional parameters needed to configure the search. For example, in time series problems, values should be passed in for the time_index, gap, forecast_horizon, and max_delay variables.
- **n_splits** (*int*) – Number of splits to use with the default data splitter.
- **timing** (*boolean*) – Whether or not to write pipeline search times to the logger. Defaults to False.
- ****kwargs** – Other keyword arguments which are provided will be passed to AutoMLSearch.

**Returns** the automl search object containing pipelines and rankings, and the results from running the data checks. If the data check results contain errors, automl search will not be run and an automl search object will not be returned.

**Return type** (*AutoMLSearch*, dict)

**Raises** **ValueError** – If the search configuration is invalid.

### callbacks

Callbacks available to pass to AutoML.

### Module Contents

### Functions

| | |
|---|---|
| *log_error_callback* | Logs the exception thrown as an error. |
| *raise_error_callback* | Raises the exception thrown by the AutoMLSearch object. |
| *silent_error_callback* | No-op. |

### Attributes Summary

| |
|---|
| *logger* |

### Contents

evalml.automl.callbacks.**log_error_callback**(*exception*, *traceback*, *automl*, ***kwargs*)

Logs the exception thrown as an error.

Will not throw. This is the default behavior for AutoMLSearch.

**Parameters**

- **exception** – Exception to log.
- **traceback** – Exception traceback to log.
- **automl** – AutoMLSearch object.
- ****kwargs** – Other relevant keyword arguments to log.

evalml.automl.callbacks.**logger**

evalml.automl.callbacks.**raise_error_callback**(*exception*, *traceback*, *automl*, ***kwargs*)

Raises the exception thrown by the AutoMLSearch object.

Also logs the exception as an error.

**Parameters**

- **exception** – Exception to log and raise.
- **traceback** – Exception traceback to log.
- **automl** – AutoMLSearch object.
- ****kwargs** – Other relevant keyword arguments to log.

**Raises** **exception** – Raises the input exception.

`evalml.automl.callbacks.silent_error_callback(exception, traceback, automl, **kwargs)`

No-op.

### pipeline_search_plots

Plots displayed during pipeline search.

### Module Contents

### Classes Summary

| | |
|---|---|
| *PipelineSearchPlots* | Plots for the AutoMLSearch class during search. |
| *SearchIterationPlot* | Search iteration plot. |

### Contents

**class** `evalml.automl.pipeline_search_plots.PipelineSearchPlots(results, objective)`

Plots for the AutoMLSearch class during search.

**Parameters**

- **results** (*dict*) – Dictionary of current results.
- **objective** (*ObjectiveBase*) – Objective that AutoML is optimizing for.

**Methods**

| | |
|---|---|
| *search_iteration_plot* | Shows a plot of the best score at each iteration using data gathered during training. |

`search_iteration_plot(self, interactive_plot=False)`

Shows a plot of the best score at each iteration using data gathered during training.

**Parameters** **interactive_plot** (*bool*) – Whether or not to show an interactive plot. Defaults to False.

**Returns** plot

**Raises** **ValueError** – If engine_str is not a valid engine.

**class** `evalml.automl.pipeline_search_plots.SearchIterationPlot(results, objective)`

Search iteration plot.

**Parameters**

- **results** (*dict*) – Dictionary of current results.
- **objective** (*ObjectiveBase*) – Objective that AutoML is optimizing for.

**Methods**

| | |
|---|---|
| *update* | Update the search plot. |

**update**(*self*, *results*, *objective*)

Update the search plot.

## progress

Progress abstraction holding stopping criteria and progress information.

### Module Contents

### Classes Summary

| | |
|---|---|
| *Progress* | Progress object holding stopping criteria and progress information. |

### Contents

**class** evalml.automl.progress.**Progress**(*max_time=None*, *max_batches=None*, *max_iterations=None*, *patience=None*, *tolerance=None*, *automl_algorithm=None*, *objective=None*, *verbose=False*)

Progress object holding stopping criteria and progress information.

**Parameters**

- **max_time** (*int*) – Maximum time to search for pipelines.
- **max_iterations** (*int*) – Maximum number of iterations to search.
- **max_batches** (*int*) – The maximum number of batches of pipelines to search. Parameters max_time, and max_iterations have precedence over stopping the search.
- **patience** (*int*) – Number of iterations without improvement to stop search early.
- **tolerance** (*float*) – Minimum percentage difference to qualify as score improvement for early stopping.
- **automl_algorithm** (*str*) – The automl algorithm to use. Used to calculate iterations if max_batches is selected as stopping criteria.
- **objective** (*str, ObjectiveBase*) – The objective used in search.
- **verbose** (*boolean*) – Whether or not to log out stopping information.

**Methods**

| | |
|---|---|
| *elapsed* | Return time elapsed using the start time and current time. |
| *return_progress* | Return information about current and end state of each stopping criteria in order of priority. |
| *should_continue* | Given AutoML Results, return whether or not the search should continue. |
| *start_timing* | Sets start time to current time. |

**elapsed**(*self*)

Return time elapsed using the start time and current time.

**return_progress**(*self*)

Return information about current and end state of each stopping criteria in order of priority.

**Returns** list of dictionaries containing information of each stopping criteria.

**Return type** List[Dict[str, unit]]

**should_continue**(*self*, *results*, *interrupted=False*, *mid_batch=False*)

Given AutoML Results, return whether or not the search should continue.

**Parameters**

- **results** (*dict*) – AutoMLSearch results.
- **interrupted** (*bool*) – whether AutoMLSearch was given an keyboard interrupt. Defaults to False.
- **mid_batch** (*bool*) – whether this method was called while in the middle of a batch or not. Defaults to False.

**Returns** True if search should continue, False otherwise.

**Return type** bool

**start_timing**(*self*)

Sets start time to current time.

### utils

Utilities useful in AutoML.

### Module Contents

### Functions

| | |
|---|---|
| *check_all_pipeline_names_unique* | Checks whether all the pipeline names are unique. |
| *get_best_sampler_for_data* | Returns the name of the sampler component to use for AutoMLSearch. |
| *get_default_primary_search_objective* | Get the default primary search objective for a problem type. |
| *get_pipelines_from_component_graphs* | Returns created pipelines from passed component graphs based on the specified problem type. |
| *get_threshold_tuning_info* | Determine for a given automl config and pipeline what the threshold tuning objective should be and whether or not training data should be further split to achieve proper threshold tuning. |
| *make_data_splitter* | Given the training data and ML problem parameters, compute a data splitting method to use during AutoML search. |
| *resplit_training_data* | Further split the training data for a given pipeline. This is needed for binary pipelines in order to properly tune the threshold. |
| *tune_binary_threshold* | Tunes the threshold of a binary pipeline to the X and y thresholding data. |

### Attributes Summary

| |
|---|
| *AutoMLConfig* |

### Contents

evalml.automl.utils.**AutoMLConfig**

evalml.automl.utils.**check_all_pipeline_names_unique**(*pipelines*)

Checks whether all the pipeline names are unique.

**Parameters** **pipelines** (*list[PipelineBase]*) – List of pipelines to check if all names are unique.

**Raises** **ValueError** – If any pipeline names are duplicated.

evalml.automl.utils.**get_best_sampler_for_data**(*X*, *y*, *sampler_method*, *sampler_balanced_ratio*)

Returns the name of the sampler component to use for AutoMLSearch.

**Parameters**

- **X** (*pd.DataFrame*) – The input feature data
- **y** (*pd.Series*) – The input target data
- **sampler_method** (*str*) – The sampler_type argument passed to AutoMLSearch

- **sampler_balanced_ratio** (*float*) – The ratio of min:majority targets that we would consider balanced, or should balance the classes to.

**Returns** The string name of the sampling component to use, or None if no sampler is necessary

**Return type** str, None

evalml.automl.utils.**get_default_primary_search_objective**(*problem_type*)

Get the default primary search objective for a problem type.

**Parameters** **problem_type** (*str or ProblemType*) – Problem type of interest.

**Returns** primary objective instance for the problem type.

**Return type** ObjectiveBase

evalml.automl.utils.**get_pipelines_from_component_graphs**(*component_graphs_dict*, *problem_type*, *parameters=None*, *random_seed=0*)

Returns created pipelines from passed component graphs based on the specified problem type.

**Parameters**

- **component_graphs_dict** (*dict*) – The dict of component graphs.
- **problem_type** (*str or ProblemType*) – The problem type for which pipelines will be created.
- **parameters** (*dict*) – Pipeline-level parameters that should be passed to the proposed pipelines. Defaults to None.
- **random_seed** (*int*) – Random seed. Defaults to 0.

**Returns** List of pipelines made from the passed component graphs.

**Return type** list

evalml.automl.utils.**get_threshold_tuning_info**(*automl_config*, *pipeline*)

Determine for a given automl config and pipeline what the threshold tuning objective should be and whether or not training data should be further split to achieve proper threshold tuning.

Can also be used after automl search has been performed to determine whether the full training data was used to train the pipeline.

**Parameters**

- **automl_config** (AutoMLConfig) – The AutoMLSearch's config object. Used to determine threshold tuning objective and whether data needs resplitting.
- **pipeline** (*Pipeline*) – The pipeline instance to Threshold.

**Returns** threshold_tuning_objective, data_needs_resplitting (str, bool)

evalml.automl.utils.**make_data_splitter**(*X*, *y*, *problem_type*, *problem_configuration=None*, *n_splits=3*, *shuffle=True*, *random_seed=0*)

Given the training data and ML problem parameters, compute a data splitting method to use during AutoML search.

**Parameters**

- **X** (*pd.DataFrame*) – The input training data of shape [n_samples, n_features].
- **y** (*pd.Series*) – The target training data of length [n_samples].
- **problem_type** (*ProblemType*) – The type of machine learning problem.

- **problem_configuration** (*dict, None*) – Additional parameters needed to configure the search. For example, in time series problems, values should be passed in for the time_index, gap, and max_delay variables. Defaults to None.
- **n_splits** (*int, None*) – The number of CV splits, if applicable. Defaults to 3.
- **shuffle** (*bool*) – Whether or not to shuffle the data before splitting, if applicable. Defaults to True.
- **random_seed** (*int*) – Seed for the random number generator. Defaults to 0.

**Returns** Data splitting method.

**Return type** sklearn.model_selection.BaseCrossValidator

**Raises** **ValueError** – If problem_configuration is not given for a time-series problem.

evalml.automl.utils.**resplit_training_data**(*pipeline*, *X_train*, *y_train*)

Further split the training data for a given pipeline. This is needed for binary pipelines in order to properly tune the threshold.

Can be used after automl search has been performed to recreate the data that was used to train a pipeline.

**Parameters**

- **pipeline** (*PipelineBase*) – the pipeline whose training data we are splitting
- **X_train** (*pd.DataFrame or np.ndarray*) – training data of shape [n_samples, n_features]
- **y_train** (*pd.Series, or np.ndarray*) – training target data of length [n_samples]

**Returns** Feature and target data each split into train and threshold tuning sets.

**Return type** pd.DataFrame, pd.DataFrame, pd.Series, pd.Series

evalml.automl.utils.**tune_binary_threshold**(*pipeline*, *objective*, *problem_type*, *X_threshold_tuning*, *y_threshold_tuning*, *X=None*, *y=None*)

Tunes the threshold of a binary pipeline to the X and y thresholding data.

**Parameters**

- **pipeline** (*Pipeline*) – Pipeline instance to threshold.
- **objective** (*ObjectiveBase*) – The objective we want to tune with. If not tuneable and best_pipeline is True, will use F1.
- **problem_type** (*ProblemType*) – The problem type of the pipeline.
- **X_threshold_tuning** (*pd.DataFrame*) – Features to which the pipeline will be tuned.
- **y_threshold_tuning** (*pd.Series*) – Target data to which the pipeline will be tuned.
- **X** (*pd.DataFrame*) – Features to which the pipeline will be trained (used for time series binary). Defaults to None.
- **y** (*pd.Series*) – Target to which the pipeline will be trained (used for time series binary). Defaults to None.

### Package Contents

### Classes Summary

| | |
|---|---|
| *AutoMLSearch* | Automated Pipeline search. |
| *EngineBase* | Base class for EvalML engines. |
| *Progress* | Progress object holding stopping criteria and progress information. |
| *SequentialEngine* | The default engine for the AutoML search. |

### Functions

| | |
|---|---|
| *get_default_primary_search_objective* | Get the default primary search objective for a problem type. |
| *get_threshold_tuning_info* | Determine for a given automl config and pipeline what the threshold tuning objective should be and whether or not training data should be further split to achieve proper threshold tuning. |
| *make_data_splitter* | Given the training data and ML problem parameters, compute a data splitting method to use during AutoML search. |
| *resplit_training_data* | Further split the training data for a given pipeline. This is needed for binary pipelines in order to properly tune the threshold. |
| *search* | Given data and configuration, run an automl search. |
| *search_iterative* | Given data and configuration, run an automl search. |
| *tune_binary_threshold* | Tunes the threshold of a binary pipeline to the X and y thresholding data. |

### Contents

**class** evalml.automl.**AutoMLSearch**(*X_train=None*, *y_train=None*, *X_holdout=None*, *y_holdout=None*, *problem_type=None*, *objective='auto'*, *max_iterations=None*, *max_time=None*, *patience=None*, *tolerance=None*, *data_splitter=None*, *allowed_component_graphs=None*, *allowed_model_families=None*, *features=None*, *start_iteration_callback=None*, *add_result_callback=None*, *error_callback=None*, *additional_objectives=None*, *alternate_thresholding_objective='F1'*, *random_seed=0*, *n_jobs=- 1*, *tuner_class=None*, *optimize_thresholds=True*, *ensembling=False*, *max_batches=None*, *problem_configuration=None*, *train_best_pipeline=True*, *search_parameters=None*, *sampler_method='auto'*, *sampler_balanced_ratio=0.25*, *allow_long_running_models=False*, *_pipelines_per_batch=5*, *automl_algorithm='default'*, *engine='sequential'*, *verbose=False*, *timing=False*, *exclude_featurizers=None*, *holdout_set_size=0*)

Automated Pipeline search.

**Parameters**

- **X_train** (*pd.DataFrame*) – The input training data of shape [n_samples, n_features]. Required.
- **y_train** (*pd.Series*) – The target training data of length [n_samples]. Required for supervised learning tasks.
- **X_holdout** (*pd.DataFrame*) – The input holdout data of shape [n_samples, n_features].
- **y_holdout** (*pd.Series*) – The target holdout data of length [n_samples].
- **problem_type** (*str or ProblemTypes*) – Type of supervised learning problem. See evalml.problem_types.ProblemType.all_problem_types for a full list.
- **objective** (*str, ObjectiveBase*) – The objective to optimize for. Used to propose and rank pipelines, but not for optimizing each pipeline during fit-time. When set to 'auto', chooses: - LogLossBinary for binary classification problems, - LogLossMulticlass for multiclass classification problems, and - R2 for regression problems.
- **max_iterations** (*int*) – Maximum number of iterations to search. If max_iterations and max_time is not set, then max_iterations will default to max_iterations of 5.
- **max_time** (*int, str*) – Maximum time to search for pipelines. This will not start a new pipeline search after the duration has elapsed. If it is an integer, then the time will be in seconds. For strings, time can be specified as seconds, minutes, or hours.
- **patience** (*int*) – Number of iterations without improvement to stop search early. Must be positive. If None, early stopping is disabled. Defaults to None.
- **tolerance** (*float*) – Minimum percentage difference to qualify as score improvement for early stopping. Only applicable if patience is not None. Defaults to None.
- **allowed_component_graphs** (*dict*) – A dictionary of lists or ComponentGraphs indicating the component graphs allowed in the search. The format should follow { "Name_0": [list_of_components], "Name_1": ComponentGraph(...) }

  The default of None indicates all pipeline component graphs for this problem type are allowed. Setting this field will cause allowed_model_families to be ignored.

  e.g. allowed_component_graphs = { "My_Graph": ["Imputer", "One Hot Encoder", "Random Forest Classifier"] }
- **allowed_model_families** (*list(str, ModelFamily)*) – The model families to search. The default of None searches over all model families. Run evalml.pipelines.components.utils.allowed_model_families("binary") to see options. Change *binary* to *multiclass* or *regression* depending on the problem type. Note that if allowed_pipelines is provided, this parameter will be ignored.
- **features** (*list*) – List of features to run DFS on AutoML pipelines. Defaults to None. Features will only be computed if the columns used by the feature exist in the search input and if the feature itself is not in search input. If features is an empty list, the DFS Transformer will not be included in pipelines.
- **data_splitter** (*sklearn.model_selection.BaseCrossValidator*) – Data splitting method to use. Defaults to StratifiedKFold.
- **tuner_class** – The tuner class to use. Defaults to SKOptTuner.
- **optimize_thresholds** (*bool*) – Whether or not to optimize the binary pipeline threshold. Defaults to True.
- **start_iteration_callback** (*callable*) – Function called before each pipeline training iteration. Callback function takes three positional parameters: The pipeline instance and the AutoMLSearch object.

- **add_result_callback** (*callable*) – Function called after each pipeline training iteration. Callback function takes three positional parameters: A dictionary containing the training results for the new pipeline, an untrained_pipeline containing the parameters used during training, and the AutoMLSearch object.
- **error_callback** (*callable*) – Function called when *search()* errors and raises an Exception. Callback function takes three positional parameters: the Exception raised, the traceback, and the AutoMLSearch object. Must also accepts kwargs, so AutoMLSearch is able to pass along other appropriate parameters by default. Defaults to None, which will call *log_error_callback*.
- **additional_objectives** (*list*) – Custom set of objectives to score on. Will override default objectives for problem type if not empty.
- **alternate_thresholding_objective** (*str*) – The objective to use for thresholding binary classification pipelines if the main objective provided isn't tuneable. Defaults to F1.
- **random_seed** (*int*) – Seed for the random number generator. Defaults to 0.
- **n_jobs** (*int or None*) – Non-negative integer describing level of parallelism used for pipelines. None and 1 are equivalent. If set to -1, all CPUs are used. For n_jobs below -1, (n_cpus + 1 + n_jobs) are used.
- **ensembling** (*boolean*) – If True, runs ensembling in a separate batch after every allowed pipeline class has been iterated over. If the number of unique pipelines to search over per batch is one, ensembling will not run. Defaults to False.
- **max_batches** (*int*) – The maximum number of batches of pipelines to search. Parameters max_time, and max_iterations have precedence over stopping the search.
- **problem_configuration** (*dict, None*) – Additional parameters needed to configure the search. For example, in time series problems, values should be passed in for the time_index, gap, forecast_horizon, and max_delay variables.
- **train_best_pipeline** (*boolean*) – Whether or not to train the best pipeline before returning it. Defaults to True.
- **search_parameters** (*dict*) – A dict of the hyperparameter ranges or pipeline parameters used to iterate over during search. Keys should consist of the component names and values should specify a singular value/list for pipeline parameters, or skopt.Space for hyperparameter ranges. In the example below, the Imputer parameters would be passed to the hyperparameter ranges, and the Label Encoder parameters would be used as the component parameter.

  **e.g. search_parameters = { 'Imputer'** [{ 'numeric_impute_strategy': Categorical(['most_frequent', 'median']) },] 'Label Encoder': {'positive_label': True} }
- **sampler_method** (*str*) – The data sampling component to use in the pipelines if the problem type is classification and the target balance is smaller than the sampler_balanced_ratio. Either 'auto', which will use our preferred sampler for the data, 'Undersampler', 'Oversampler', or None. Defaults to 'auto'.
- **sampler_balanced_ratio** (*float*) – The minority:majority class ratio that we consider balanced, so a 1:4 ratio would be equal to 0.25. If the class balance is larger than this provided value, then we will not add a sampler since the data is then considered balanced. Overrides the *sampler_ratio* of the samplers. Defaults to 0.25.
- **allow_long_running_models** (*bool*) – Whether or not to allow longer-running models for large multiclass problems. If False and no pipelines, component graphs, or model families are provided, AutoMLSearch will not use Elastic Net or XGBoost when there are more than

75 multiclass targets and will not use CatBoost when there are more than 150 multiclass targets. Defaults to False.

- **_ensembling_split_size** (*float*) – The amount of the training data we'll set aside for training ensemble metalearners. Only used when ensembling is True. Must be between 0 and 1, exclusive. Defaults to 0.2
- **_pipelines_per_batch** (*int*) – The number of pipelines to train for every batch after the first one. The first batch will train a baseline pipline + one of each pipeline family allowed in the search.
- **automl_algorithm** (*str*) – The automl algorithm to use. Currently the two choices are 'iterative' and 'default'. Defaults to *default*.
- **engine** (EngineBase *or str*) – The engine instance used to evaluate pipelines. Dask or concurrent.futures engines can also be chosen by providing a string from the list ["sequential", "cf_threaded", "cf_process", "dask_threaded", "dask_process"]. If a parallel engine is selected this way, the maximum amount of parallelism, as determined by the engine, will be used. Defaults to "sequential".
- **verbose** (*boolean*) – Whether or not to display semi-real-time updates to stdout while search is running. Defaults to False.
- **timing** (*boolean*) – Whether or not to write pipeline search times to the logger. Defaults to False.
- **exclude_featurizers** (*list[str]*) – A list of featurizer components to exclude from the pipelines built by search. Valid options are "DatetimeFeaturizer", "EmailFeaturizer", "URLFeaturizer", "NaturalLanguageFeaturizer", "TimeSeriesFeaturizer"
- **holdout_set_size** (*float*) – The size of the holdout set that AutoML search will take for datasets larger than 500 rows. If set to 0, holdout set will not be taken regardless of number of rows. Must be between 0 and 1, exclusive. Defaults to 0.1.

**Methods**

| | |
|---|---|
| *add_to_rankings* | Fits and evaluates a given pipeline then adds the results to the automl rankings with the requirement that automl search has been run. |
| *best_pipeline* | Returns a trained instance of the best pipeline and parameters found during automl search. If *train_best_pipeline* is set to False, returns an untrained pipeline instance. |
| *close_engine* | Function to explicitly close the engine, client, parallel resources. |
| *describe_pipeline* | Describe a pipeline. |
| *full_rankings* | Returns a pandas.DataFrame with scoring results from all pipelines searched. |
| *get_ensembler_input_pipelines* | Returns a list of input pipeline IDs given an ensembler pipeline ID. |
| *get_pipeline* | Given the ID of a pipeline training result, returns an untrained instance of the specified pipeline initialized with the parameters used to train that pipeline during automl search. |
| *load* | Loads AutoML object at file path. |
| *plot* | Return an instance of the plot with the latest scores. |
| *rankings* | Returns a pandas.DataFrame with scoring results from the highest-scoring set of parameters used with each pipeline. |
| *results* | Class that allows access to a copy of the results from *automl_search*. |
| *save* | Saves AutoML object at file path. |
| *score_pipelines* | Score a list of pipelines on the given holdout data. |
| *search* | Find the best pipeline for the data set. |
| *train_pipelines* | Train a list of pipelines on the training data. |

**add_to_rankings**(*self*, *pipeline*)

Fits and evaluates a given pipeline then adds the results to the automl rankings with the requirement that automl search has been run.

**Parameters** **pipeline** (*PipelineBase*) – pipeline to train and evaluate.

**property best_pipeline**(*self*)

Returns a trained instance of the best pipeline and parameters found during automl search. If *train_best_pipeline* is set to False, returns an untrained pipeline instance.

**Returns** A trained instance of the best pipeline and parameters found during automl search. If *train_best_pipeline* is set to False, returns an untrained pipeline instance.

**Return type** PipelineBase

**Raises** **PipelineNotFoundError** – If this is called before .search() is called.

**close_engine**(*self*)

Function to explicitly close the engine, client, parallel resources.

**describe_pipeline**(*self*, *pipeline_id*, *return_dict=False*)

Describe a pipeline.

**Parameters**

- **pipeline_id** (*int*) – pipeline to describe

- **return_dict** (*bool*) – If True, return dictionary of information about pipeline. Defaults to False.

**Returns** Description of specified pipeline. Includes information such as type of pipeline components, problem, training time, cross validation, etc.

**Raises** **PipelineNotFoundError** – If pipeline_id is not a valid ID.

**property full_rankings**(*self*)

Returns a pandas.DataFrame with scoring results from all pipelines searched.

**get_ensembler_input_pipelines**(*self*, *ensemble_pipeline_id*)

Returns a list of input pipeline IDs given an ensembler pipeline ID.

**Parameters** **ensemble_pipeline_id** (*id*) – Ensemble pipeline ID to get input pipeline IDs from.

**Returns** A list of ensemble input pipeline IDs.

**Return type** list[int]

**Raises** **ValueError** – If *ensemble_pipeline_id* does not correspond to a valid ensemble pipeline ID.

**get_pipeline**(*self*, *pipeline_id*)

Given the ID of a pipeline training result, returns an untrained instance of the specified pipeline initialized with the parameters used to train that pipeline during automl search.

**Parameters** **pipeline_id** (*int*) – Pipeline to retrieve.

**Returns** Untrained pipeline instance associated with the provided ID.

**Return type** PipelineBase

**Raises** **PipelineNotFoundError** – if pipeline_id is not a valid ID.

**static load**(*file_path*, *pickle_type='cloudpickle'*)

Loads AutoML object at file path.

**Parameters**

- **file_path** (*str*) – Location to find file to load
- **pickle_type** (*{"pickle", "cloudpickle"}*) – The pickling library to use. Currently not used since the standard pickle library can handle cloudpickles.

**Returns** AutoSearchBase object

**property plot**(*self*)

Return an instance of the plot with the latest scores.

**property rankings**(*self*)

Returns a pandas.DataFrame with scoring results from the highest-scoring set of parameters used with each pipeline.

**property results**(*self*)

Class that allows access to a copy of the results from *automl_search*.

**Returns**

**Dictionary containing *pipeline_results*, a dict with results from each pipeline,** and *search_order*, a list describing the order the pipelines were searched.

**Return type** dict

**save**(*self*, *file_path*, *pickle_type='cloudpickle'*, *pickle_protocol=cloudpickle.DEFAULT_PROTOCOL*)

Saves AutoML object at file path.

**Parameters**

- **file_path** (*str*) – Location to save file.
- **pickle_type** (*{"pickle", "cloudpickle"}*) – The pickling library to use.
- **pickle_protocol** (*int*) – The pickle data stream format.

**Raises** **ValueError** – If pickle_type is not "pickle" or "cloudpickle".

**score_pipelines**(*self*, *pipelines*, *X_holdout*, *y_holdout*, *objectives*)

Score a list of pipelines on the given holdout data.

**Parameters**

- **pipelines** (*list[PipelineBase]*) – List of pipelines to train.
- **X_holdout** (*pd.DataFrame*) – Holdout features.
- **y_holdout** (*pd.Series*) – Holdout targets for scoring.
- **objectives** (*list[str], list[ObjectiveBase]*) – Objectives used for scoring.

**Returns** Dictionary keyed by pipeline name that maps to a dictionary of scores. Note that the any pipelines that error out during scoring will not be included in the dictionary but the exception and stacktrace will be displayed in the log.

**Return type** dict[str, Dict[str, float]]

**search**(*self*, *interactive_plot=True*)

Find the best pipeline for the data set.

**Parameters** **interactive_plot** (*boolean, True*) – Shows an iteration vs. score plot in Jupyter notebook. Disabled by default in non-Jupyter enviroments.

**Raises** **AutoMLSearchException** – If all pipelines in the current AutoML batch produced a score of np.nan on the primary objective.

**Returns** Dictionary keyed by batch number that maps to the timings for pipelines run in that batch, as well as the total time for each batch. Pipelines within a batch are labeled by pipeline name.

**Return type** Dict[int, Dict[str, Timestamp]]

**train_pipelines**(*self*, *pipelines*)

Train a list of pipelines on the training data.

This can be helpful for training pipelines once the search is complete.

**Parameters** **pipelines** (*list[PipelineBase]*) – List of pipelines to train.

**Returns** Dictionary keyed by pipeline name that maps to the fitted pipeline. Note that the any pipelines that error out during training will not be included in the dictionary but the exception and stacktrace will be displayed in the log.

**Return type** Dict[str, PipelineBase]

**class** evalml.automl.**EngineBase**

Base class for EvalML engines.

**Methods**

| | |
|---|---|
| *setup_job_log* | Set up logger for job. |
| *submit_evaluation_job* | Submit job for pipeline evaluation during AutoMLSearch. |
| *submit_scoring_job* | Submit job for pipeline scoring. |
| *submit_training_job* | Submit job for pipeline training. |

**static setup_job_log**()

Set up logger for job.

**abstract submit_evaluation_job**(*self*, *automl_config*, *pipeline*, *X*, *y*, *X_holdout=None*, *y_holdout=None*)

Submit job for pipeline evaluation during AutoMLSearch.

**abstract submit_scoring_job**(*self*, *automl_config*, *pipeline*, *X*, *y*, *objectives*, *X_train=None*, *y_train=None*)

Submit job for pipeline scoring.

**abstract submit_training_job**(*self*, *automl_config*, *pipeline*, *X*, *y*, *X_holdout=None*, *y_holdout=None*)

Submit job for pipeline training.

evalml.automl.**get_default_primary_search_objective**(*problem_type*)

Get the default primary search objective for a problem type.

**Parameters** **problem_type** (*str or ProblemType*) – Problem type of interest.

**Returns** primary objective instance for the problem type.

**Return type** ObjectiveBase

evalml.automl.**get_threshold_tuning_info**(*automl_config*, *pipeline*)

Determine for a given automl config and pipeline what the threshold tuning objective should be and whether or not training data should be further split to achieve proper threshold tuning.

Can also be used after automl search has been performed to determine whether the full training data was used to train the pipeline.

**Parameters**

- **automl_config** (*AutoMLConfig*) – The AutoMLSearch's config object. Used to determine threshold tuning objective and whether data needs resplitting.
- **pipeline** (*Pipeline*) – The pipeline instance to Threshold.

**Returns** threshold_tuning_objective, data_needs_resplitting (str, bool)

evalml.automl.**make_data_splitter**(*X*, *y*, *problem_type*, *problem_configuration=None*, *n_splits=3*, *shuffle=True*, *random_seed=0*)

Given the training data and ML problem parameters, compute a data splitting method to use during AutoML search.

**Parameters**

- **X** (*pd.DataFrame*) – The input training data of shape [n_samples, n_features].
- **y** (*pd.Series*) – The target training data of length [n_samples].
- **problem_type** (*ProblemType*) – The type of machine learning problem.
- **problem_configuration** (*dict, None*) – Additional parameters needed to configure the search. For example, in time series problems, values should be passed in for the time_index, gap, and max_delay variables. Defaults to None.

- **n_splits** (*int, None*) – The number of CV splits, if applicable. Defaults to 3.
- **shuffle** (*bool*) – Whether or not to shuffle the data before splitting, if applicable. Defaults to True.
- **random_seed** (*int*) – Seed for the random number generator. Defaults to 0.

**Returns** Data splitting method.

**Return type** sklearn.model_selection.BaseCrossValidator

**Raises** **ValueError** – If problem_configuration is not given for a time-series problem.

**class** evalml.automl.**Progress**(*max_time=None*, *max_batches=None*, *max_iterations=None*, *patience=None*, *tolerance=None*, *automl_algorithm=None*, *objective=None*, *verbose=False*)

Progress object holding stopping criteria and progress information.

**Parameters**

- **max_time** (*int*) – Maximum time to search for pipelines.
- **max_iterations** (*int*) – Maximum number of iterations to search.
- **max_batches** (*int*) – The maximum number of batches of pipelines to search. Parameters max_time, and max_iterations have precedence over stopping the search.
- **patience** (*int*) – Number of iterations without improvement to stop search early.
- **tolerance** (*float*) – Minimum percentage difference to qualify as score improvement for early stopping.
- **automl_algorithm** (*str*) – The automl algorithm to use. Used to calculate iterations if max_batches is selected as stopping criteria.
- **objective** (*str, ObjectiveBase*) – The objective used in search.
- **verbose** (*boolean*) – Whether or not to log out stopping information.

**Methods**

| | |
|---|---|
| *elapsed* | Return time elapsed using the start time and current time. |
| *return_progress* | Return information about current and end state of each stopping criteria in order of priority. |
| *should_continue* | Given AutoML Results, return whether or not the search should continue. |
| *start_timing* | Sets start time to current time. |

**elapsed**(*self*)

Return time elapsed using the start time and current time.

**return_progress**(*self*)

Return information about current and end state of each stopping criteria in order of priority.

**Returns** list of dictionaries containing information of each stopping criteria.

**Return type** List[Dict[str, unit]]

**should_continue**(*self*, *results*, *interrupted=False*, *mid_batch=False*)

Given AutoML Results, return whether or not the search should continue.

**Parameters**

- **results** (*dict*) – AutoMLSearch results.
- **interrupted** (*bool*) – whether AutoMLSearch was given an keyboard interrupt. Defaults to False.
- **mid_batch** (*bool*) – whether this method was called while in the middle of a batch or not. Defaults to False.

**Returns** True if search should continue, False otherwise.

**Return type** bool

**start_timing**(*self*)

Sets start time to current time.

evalml.automl.**resplit_training_data**(*pipeline*, *X_train*, *y_train*)

Further split the training data for a given pipeline. This is needed for binary pipelines in order to properly tune the threshold.

Can be used after automl search has been performed to recreate the data that was used to train a pipeline.

**Parameters**

- **pipeline** (*PipelineBase*) – the pipeline whose training data we are splitting
- **X_train** (*pd.DataFrame or np.ndarray*) – training data of shape [n_samples, n_features]
- **y_train** (*pd.Series, or np.ndarray*) – training target data of length [n_samples]

**Returns** Feature and target data each split into train and threshold tuning sets.

**Return type** pd.DataFrame, pd.DataFrame, pd.Series, pd.Series

evalml.automl.**search**(*X_train=None*, *y_train=None*, *problem_type=None*, *objective='auto'*, *mode='fast'*, *max_time=None*, *patience=None*, *tolerance=None*, *problem_configuration=None*, *n_splits=3*, *verbose=False*, *timing=False*)

Given data and configuration, run an automl search.

This method will run EvalML's default suite of data checks. If the data checks produce errors, the data check results will be returned before running the automl search. In that case we recommend you alter your data to address these errors and try again. This method is provided for convenience. If you'd like more control over when each of these steps is run, consider making calls directly to the various pieces like the data checks and AutoMLSearch, instead of using this method.

**Parameters**

- **X_train** (*pd.DataFrame*) – The input training data of shape [n_samples, n_features]. Required.
- **y_train** (*pd.Series*) – The target training data of length [n_samples]. Required for supervised learning tasks.
- **problem_type** (*str or ProblemTypes*) – Type of supervised learning problem. See evalml.problem_types.ProblemType.all_problem_types for a full list.
- **objective** (*str, ObjectiveBase*) – The objective to optimize for. Used to propose and rank pipelines, but not for optimizing each pipeline during fit-time. When set to 'auto', chooses: - LogLossBinary for binary classification problems, - LogLossMulticlass for multiclass classification problems, and - R2 for regression problems.
- **mode** (*str*) – mode for DefaultAlgorithm. There are two modes: fast and long, where fast is a subset of long. Please look at DefaultAlgorithm for more details.

- **max_time** (*int, str*) – Maximum time to search for pipelines. This will not start a new pipeline search after the duration has elapsed. If it is an integer, then the time will be in seconds. For strings, time can be specified as seconds, minutes, or hours.
- **patience** (*int*) – Number of iterations without improvement to stop search early. Must be positive. If None, early stopping is disabled. Defaults to None.
- **tolerance** (*float*) – Minimum percentage difference to qualify as score improvement for early stopping. Only applicable if patience is not None. Defaults to None.
- **problem_configuration** (*dict*) – Additional parameters needed to configure the search. For example, in time series problems, values should be passed in for the time_index, gap, forecast_horizon, and max_delay variables.
- **n_splits** (*int*) – Number of splits to use with the default data splitter.
- **verbose** (*boolean*) – Whether or not to display semi-real-time updates to stdout while search is running. Defaults to False.
- **timing** (*boolean*) – Whether or not to write pipeline search times to the logger. Defaults to False.

**Returns** The automl search object containing pipelines and rankings, and the results from running the data checks. If the data check results contain errors, automl search will not be run and an automl search object will not be returned.

**Return type** (*AutoMLSearch*, dict)

**Raises** **ValueError** – If search configuration is not valid.

evalml.automl.**search_iterative**(*X_train=None*, *y_train=None*, *problem_type=None*, *objective='auto'*, *problem_configuration=None*, *n_splits=3*, *timing=False*, ***kwargs*)

Given data and configuration, run an automl search.

This method will run EvalML's default suite of data checks. If the data checks produce errors, the data check results will be returned before running the automl search. In that case we recommend you alter your data to address these errors and try again. This method is provided for convenience. If you'd like more control over when each of these steps is run, consider making calls directly to the various pieces like the data checks and AutoMLSearch, instead of using this method.

**Parameters**

- **X_train** (*pd.DataFrame*) – The input training data of shape [n_samples, n_features]. Required.
- **y_train** (*pd.Series*) – The target training data of length [n_samples]. Required for supervised learning tasks.
- **problem_type** (*str or ProblemTypes*) – Type of supervised learning problem. See evalml.problem_types.ProblemType.all_problem_types for a full list.
- **objective** (*str, ObjectiveBase*) – The objective to optimize for. Used to propose and rank pipelines, but not for optimizing each pipeline during fit-time. When set to 'auto', chooses: - LogLossBinary for binary classification problems, - LogLossMulticlass for multiclass classification problems, and - R2 for regression problems.
- **problem_configuration** (*dict*) – Additional parameters needed to configure the search. For example, in time series problems, values should be passed in for the time_index, gap, forecast_horizon, and max_delay variables.
- **n_splits** (*int*) – Number of splits to use with the default data splitter.

- **timing** (*boolean*) – Whether or not to write pipeline search times to the logger. Defaults to False.
- ****kwargs** – Other keyword arguments which are provided will be passed to AutoMLSearch.

**Returns** the automl search object containing pipelines and rankings, and the results from running the data checks. If the data check results contain errors, automl search will not be run and an automl search object will not be returned.

**Return type** (*AutoMLSearch*, dict)

**Raises** **ValueError** – If the search configuration is invalid.

**class** evalml.automl.**SequentialEngine**

The default engine for the AutoML search.

Trains and scores pipelines locally and sequentially.

**Methods**

| | |
|---|---|
| *close* | No-op. |
| *setup_job_log* | Set up logger for job. |
| *submit_evaluation_job* | Submit a job to evaluate a pipeline. |
| *submit_scoring_job* | Submit a job to score a pipeline. |
| *submit_training_job* | Submit a job to train a pipeline. |

**close**(*self*)

No-op.

**static setup_job_log**()

Set up logger for job.

**submit_evaluation_job**(*self*, *automl_config*, *pipeline*, *X*, *y*, *X_holdout=None*, *y_holdout=None*)

Submit a job to evaluate a pipeline.

**Parameters**

- **automl_config** – Structure containing data passed from AutoMLSearch instance.
- **pipeline** (*pipeline.PipelineBase*) – Pipeline to evaluate.
- **X** (*pd.DataFrame*) – Input data for modeling.
- **y** (*pd.Series*) – Target data for modeling.
- **X_holdout** (*pd.Series*) – Holdout input data for holdout scoring.
- **y_holdout** (*pd.Series*) – Holdout target data for holdout scoring.

**Returns** Computation result.

**Return type** SequentialComputation

**submit_scoring_job**(*self*, *automl_config*, *pipeline*, *X*, *y*, *objectives*, *X_train=None*, *y_train=None*)

Submit a job to score a pipeline.

**Parameters**

- **automl_config** – Structure containing data passed from AutoMLSearch instance.
- **pipeline** (*pipeline.PipelineBase*) – Pipeline to train.
- **X** (*pd.DataFrame*) – Input data for modeling.

- **y** (*pd.Series*) – Target data for modeling.
- **X_train** (*pd.DataFrame*) – Training features. Used for feature engineering in time series.
- **y_train** (*pd.Series*) – Training target. Used for feature engineering in time series.
- **objectives** (*list[ObjectiveBase]*) – List of objectives to score on.

**Returns** Computation result.

**Return type** SequentialComputation

**submit_training_job**(*self*, *automl_config*, *pipeline*, *X*, *y*)

Submit a job to train a pipeline.

**Parameters**

- **automl_config** – Structure containing data passed from AutoMLSearch instance.
- **pipeline** (*pipeline.PipelineBase*) – Pipeline to evaluate.
- **X** (*pd.DataFrame*) – Input data for modeling.
- **y** (*pd.Series*) – Target data for modeling.

**Returns** Computation result.

**Return type** SequentialComputation

evalml.automl.**tune_binary_threshold**(*pipeline*, *objective*, *problem_type*, *X_threshold_tuning*, *y_threshold_tuning*, *X=None*, *y=None*)

Tunes the threshold of a binary pipeline to the X and y thresholding data.

**Parameters**

- **pipeline** (*Pipeline*) – Pipeline instance to threshold.
- **objective** (*ObjectiveBase*) – The objective we want to tune with. If not tuneable and best_pipeline is True, will use F1.
- **problem_type** (*ProblemType*) – The problem type of the pipeline.
- **X_threshold_tuning** (*pd.DataFrame*) – Features to which the pipeline will be tuned.
- **y_threshold_tuning** (*pd.Series*) – Target data to which the pipeline will be tuned.
- **X** (*pd.DataFrame*) – Features to which the pipeline will be trained (used for time series binary). Defaults to None.
- **y** (*pd.Series*) – Target to which the pipeline will be trained (used for time series binary). Defaults to None.

### Data Checks

Data checks.

### Submodules

### class_imbalance_data_check

Data check that checks if any of the target labels are imbalanced, or if the number of values for each target are below 2 times the number of CV folds.

Use for classification problems.

### Module Contents

### Classes Summary

| | |
|---|---|
| `ClassImbalanceDataCheck` | Check if any of the target labels are imbalanced, or if the number of values for each target are below 2 times the number of CV folds. Use for classification problems. |

### Contents

**class** `evalml.data_checks.class_imbalance_data_check.`**`ClassImbalanceDataCheck`**(*threshold=0.1*, *min_samples=100*, *num_cv_folds=3*, *test_size=None*)

Check if any of the target labels are imbalanced, or if the number of values for each target are below 2 times the number of CV folds. Use for classification problems.

**Parameters**

- **threshold** (*float*) – The minimum threshold allowed for class imbalance before a warning is raised. This threshold is calculated by comparing the number of samples in each class to the sum of samples in that class and the majority class. For example, a multiclass case with [900, 900, 100] samples per classes 0, 1, and 2, respectively, would have a 0.10 threshold for class 2 (100 / (900 + 100)). Defaults to 0.10.
- **min_samples** (*int*) – The minimum number of samples per accepted class. If the minority class is both below the threshold and min_samples, then we consider this severely imbalanced. Must be greater than 0. Defaults to 100.
- **num_cv_folds** (*int*) – The number of cross-validation folds. Must be positive. Choose 0 to ignore this warning. Defaults to 3.
- **test_size** (*None, float, int*) – Percentage of test set size. Used to calculate class imbalance prior to splitting the data into training and validation/test sets.

**Raises**

- **ValueError** – If threshold is not within 0 and 0.5
- **ValueError** – If min_samples is not greater than 0
- **ValueError** – If number of cv folds is negative
- **ValueError** – If test_size is not between 0 and 1

**Methods**

| | |
|---|---|
| *name* | Return a name describing the data check. |
| *validate* | Check if any target labels are imbalanced beyond a threshold for binary and multiclass problems. |

**name**(*cls*)

Return a name describing the data check.

**validate**(*self*, *X*, *y*)

Check if any target labels are imbalanced beyond a threshold for binary and multiclass problems.

Ignores NaN values in target labels if they appear.

**Parameters**

- **X** (*pd.DataFrame, np.ndarray*) – Features. Ignored.
- **y** (*pd.Series, np.ndarray*) – Target labels to check for imbalanced data.

**Returns**

**Dictionary with DataCheckWarnings if imbalance in classes is less than the threshold,** and DataCheckErrors if the number of values for each target is below 2 * num_cv_folds.

**Return type** dict

### Examples

```
>>> import pandas as pd
...
>>> X = pd.DataFrame()
>>> y = pd.Series([0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1])
```

In this binary example, the target class 0 is present in fewer than 10% (threshold=0.10) of instances, and fewer than 2 * the number of cross folds (2 * 3 = 6). Therefore, both a warning and an error are returned as part of the Class Imbalance Data Check. In addition, if a target is present with fewer than *min_samples* occurrences (default is 100) and is under the threshold, a severe class imbalance warning will be raised.

```
>>> class_imb_dc = ClassImbalanceDataCheck(threshold=0.10)
>>> assert class_imb_dc.validate(X, y) == [
...     {
...         "message": "The number of instances of these targets is less than 2␣
↪* the number of cross folds = 6 instances: [0]",
...         "data_check_name": "ClassImbalanceDataCheck",
...         "level": "error",
...         "code": "CLASS_IMBALANCE_BELOW_FOLDS",
...         "details": {"target_values": [0], "rows": None, "columns": None},
...         "action_options": []
...     },
...     {
...         "message": "The following labels fall below 10% of the target: [0]",
...         "data_check_name": "ClassImbalanceDataCheck",
...         "level": "warning",
...         "code": "CLASS_IMBALANCE_BELOW_THRESHOLD",
```

(continues on next page)

(continued from previous page)

```
...         "details": {"target_values": [0], "rows": None, "columns": None},
...         "action_options": []
...     },
...     {
...         "message": "The following labels in the target have severe class␣
↪imbalance because they fall under 10% of the target and have less than 100␣
↪samples: [0]",
...         "data_check_name": "ClassImbalanceDataCheck",
...         "level": "warning",
...         "code": "CLASS_IMBALANCE_SEVERE",
...         "details": {"target_values": [0], "rows": None, "columns": None},
...         "action_options": []
...     }
... ]
```

In this multiclass example, the target class 0 is present in fewer than 30% of observations, however with 1 cv fold, the minimum number of instances required is 2 * 1 = 2. Therefore a warning, but not an error, is raised.

```
>>> y = pd.Series([0, 0, 1, 1, 1, 1, 2, 2, 2, 2, 2])
>>> class_imb_dc = ClassImbalanceDataCheck(threshold=0.30, min_samples=5, num_
↪cv_folds=1)
>>> assert class_imb_dc.validate(X, y) == [
...     {
...         "message": "The following labels fall below 30% of the target: [0]",
...         "data_check_name": "ClassImbalanceDataCheck",
...         "level": "warning",
...         "code": "CLASS_IMBALANCE_BELOW_THRESHOLD",
...         "details": {"target_values": [0], "rows": None, "columns": None},
...         "action_options": []
...     },
...     {
...         "message": "The following labels in the target have severe class␣
↪imbalance because they fall under 30% of the target and have less than 5␣
↪samples: [0]",
...         "data_check_name": "ClassImbalanceDataCheck",
...         "level": "warning",
...         "code": "CLASS_IMBALANCE_SEVERE",
...         "details": {"target_values": [0], "rows": None, "columns": None},
...         "action_options": []
...     }
... ]
...
>>> y = pd.Series([0, 0, 1, 1, 1, 1, 2, 2, 2, 2])
>>> class_imb_dc = ClassImbalanceDataCheck(threshold=0.30, num_cv_folds=1)
>>> assert class_imb_dc.validate(X, y) == []
```

### data_check

Base class for all data checks.

### Module Contents

### Classes Summary

| | |
|---|---|
| *DataCheck* | Base class for all data checks. |

### Contents

**class** evalml.data_checks.data_check.**DataCheck**

Base class for all data checks.

Data checks are a set of heuristics used to determine if there are problems with input data.

**Methods**

| | |
|---|---|
| *name* | Return a name describing the data check. |
| *validate* | Inspect and validate the input data, runs any necessary calculations or algorithms, and returns a list of warnings and errors if applicable. |

**name**(*cls*)

Return a name describing the data check.

**abstract validate**(*self*, *X*, *y=None*)

Inspect and validate the input data, runs any necessary calculations or algorithms, and returns a list of warnings and errors if applicable.

**Parameters**

- **X** (*pd.DataFrame*) – The input data of shape [n_samples, n_features]
- **y** (*pd.Series, optional*) – The target data of length [n_samples]

**Returns** Dictionary of DataCheckError and DataCheckWarning messages

**Return type** dict (DataCheckMessage)

### data_check_action

Recommended action returned by a DataCheck.

### Module Contents

### Classes Summary

| | |
|---|---|
| *DataCheckAction* | A recommended action returned by a DataCheck. |

### Contents

**class** evalml.data_checks.data_check_action.**DataCheckAction**(*action_code*, *data_check_name*, *metadata=None*)

A recommended action returned by a DataCheck.

**Parameters**

- **action_code** (*str, DataCheckActionCode*) – Action code associated with the action.
- **data_check_name** (*str*) – Name of data check.
- **metadata** (*dict, optional*) – Additional useful information associated with the action. Defaults to None.

**Methods**

| | |
|---|---|
| *convert_dict_to_action* | Convert a dictionary into a DataCheckAction. |
| *to_dict* | Return a dictionary form of the data check action. |

**static convert_dict_to_action**(*action_dict*)

Convert a dictionary into a DataCheckAction.

**Parameters** **action_dict** – Dictionary to convert into action. Should have keys "code", "data_check_name", and "metadata".

**Raises** **ValueError** – If input dictionary does not have keys *code* and *metadata* and if the *metadata* dictionary does not have keys *columns* and *rows*.

**Returns** DataCheckAction object from the input dictionary.

**to_dict**(*self*)

Return a dictionary form of the data check action.

### data_check_action_code

Enum for data check action code.

### Module Contents

### Classes Summary

| | |
|---|---|
| *DataCheckActionCode* | Enum for data check action code. |

### Contents

**class** evalml.data_checks.data_check_action_code.**DataCheckActionCode**

Enum for data check action code.

**Attributes**

| | |
|---|---|
| **DROP_COL** | Action code for dropping a column. |
| **DROP_ROWS** | Action code for dropping rows. |
| **IMPUTE_COL** | Action code for imputing a column. |
| **REGULARIZE_AND_IMPUTE_DATASET** | Action code for regularizing and imputing all features and target time series data. |
| **SET_FIRST_COL_ID** | Action code for setting the first column as an id column. |
| **TRANSFORM_TARGET** | Action code for transforming the target data. |

**Methods**

| | |
|---|---|
| *name* | The name of the Enum member. |
| *value* | The value of the Enum member. |

**name**(*self*)

The name of the Enum member.

**value**(*self*)

The value of the Enum member.

## data_check_action_option

Recommended action returned by a DataCheck.

### Module Contents

### Classes Summary

| | |
|---|---|
| *DataCheckActionOption* | A recommended action option returned by a DataCheck. |
| *DCAOParameterAllowedValuesType* | Enum for data check action option parameter allowed values type. |
| *DCAOParameterType* | Enum for data check action option parameter type. |

## Contents

**class** evalml.data_checks.data_check_action_option.**DataCheckActionOption**(*action_code*, *data_check_name*, *parameters=None*, *metadata=None*)

A recommended action option returned by a DataCheck.

It contains an action code that indicates what the action should be, a data check name that indicates what data check was used to generate the action, and parameters and metadata which can be used to further refine the action.

**Parameters**

- **action_code** (*DataCheckActionCode*) – Action code associated with the action option.
- **data_check_name** (*str*) – Name of the data check that produced this option.
- **parameters** (*dict*) – Parameters associated with the action option. Defaults to None.
- **metadata** (*dict, optional*) – Additional useful information associated with the action option. Defaults to None.

### Examples

```
>>> parameters = {
...     "global_parameter_name": {
...         "parameter_type": "global",
...         "type": "float",
...         "default_value": 0.0,
...     },
...     "column_parameter_name": {
...         "parameter_type": "column",
...         "columns": {
...             "a": {
...                 "impute_strategy": {
...                     "categories": ["mean", "most_frequent"],
...                     "type": "category",
...                     "default_value": "mean",
...                 },
...             "constant_fill_value": {"type": "float", "default_value": 0},
...             },
...         },
...     },
... }
>>> data_check_action = DataCheckActionOption(DataCheckActionCode.DROP_COL, None,␣
↪metadata={}, parameters=parameters)
```

**Methods**

| | |
|---|---|
| *convert_dict_to_option* | Convert a dictionary into a DataCheckActionOption. |
| *get_action_from_defaults* | Returns an action based on the defaults parameters. |
| *to_dict* | Return a dictionary form of the data check action option. |

**static convert_dict_to_option**(*action_dict*)

Convert a dictionary into a DataCheckActionOption.

**Parameters** **action_dict** – Dictionary to convert into an action option. Should have keys "code", "data_check_name", and "metadata".

**Raises** **ValueError** – If input dictionary does not have keys *code* and *metadata* and if the *metadata* dictionary does not have keys *columns* and *rows*.

**Returns** DataCheckActionOption object from the input dictionary.

**get_action_from_defaults**(*self*)

Returns an action based on the defaults parameters.

**Returns** An based on the defaults parameters the option.

**Return type** DataCheckAction

**to_dict**(*self*)

Return a dictionary form of the data check action option.

**class** evalml.data_checks.data_check_action_option.**DCAOParameterAllowedValuesType**

Enum for data check action option parameter allowed values type.

**Attributes**

| | |
|---|---|
| **CATEGORICAL** | Categorical allowed values type. Parameters that have a set of allowed values. |
| **NUMERICAL** | Numerical allowed values type. Parameters that have a range of allowed values. |

**Methods**

| | |
|---|---|
| *name* | The name of the Enum member. |
| *value* | The value of the Enum member. |

**name**(*self*)

The name of the Enum member.

**value**(*self*)

The value of the Enum member.

**class** evalml.data_checks.data_check_action_option.**DCAOParameterType**

Enum for data check action option parameter type.

**Attributes**

| | |
|---|---|
| **COLUMN** | Column parameter type. Parameters that apply to a specific column in the data set. |
| **GLOBAL** | Global parameter type. Parameters that apply to the entire data set. |

**Methods**

| | |
|---|---|
| *all_parameter_types* | Get a list of all defined parameter types. |
| *handle_dcao_parameter_type* | Handles the data check action option parameter type by either returning the DCAOParameterType enum or converting from a str. |
| *name* | The name of the Enum member. |
| *value* | The value of the Enum member. |

**all_parameter_types**(*cls*)

Get a list of all defined parameter types.

**Returns** List of all defined parameter types.

**Return type** list(*DCAOParameterType*)

**static handle_dcao_parameter_type**(*dcao_parameter_type*)

Handles the data check action option parameter type by either returning the DCAOParameterType enum or converting from a str.

**Parameters** **dcao_parameter_type** (*str or DCAOParameterType*) – Data check action option parameter type that needs to be handled.

**Returns** DCAOParameterType enum

**Raises**

- **KeyError** – If input is not a valid DCAOParameterType enum value.
- **ValueError** – If input is not a string or DCAOParameterType object.

**name**(*self*)

The name of the Enum member.

**value**(*self*)

The value of the Enum member.

## data_check_message

Messages returned by a DataCheck, tagged by name.

### Module Contents

### Classes Summary

| | |
|---|---|
| *DataCheckError* | DataCheckMessage subclass for errors returned by data checks. |
| *DataCheckMessage* | Base class for a message returned by a DataCheck, tagged by name. |
| *DataCheckWarning* | DataCheckMessage subclass for warnings returned by data checks. |

## Contents

**class** evalml.data_checks.data_check_message.**DataCheckError**(*message*, *data_check_name*, *message_code=None*, *details=None*, *action_options=None*)

DataCheckMessage subclass for errors returned by data checks.

**Attributes**

| | |
|---|---|
| **message_type** | DataCheckMessageType.ERROR |

**Methods**

| | |
|---|---|
| *to_dict* | Return a dictionary form of the data check message. |

**to_dict**(*self*)

Return a dictionary form of the data check message.

**class** evalml.data_checks.data_check_message.**DataCheckMessage**(*message*, *data_check_name*, *message_code=None*, *details=None*, *action_options=None*)

Base class for a message returned by a DataCheck, tagged by name.

**Parameters**

- **message** (*str*) – Message string.
- **data_check_name** (*str*) – Name of the associated data check.
- **message_code** (*DataCheckMessageCode, optional*) – Message code associated with the message. Defaults to None.
- **details** (*dict, optional*) – Additional useful information associated with the message. Defaults to None.
- **action_options** (*list, optional*) – A list of `DataCheckActionOption`s associated with the message. Defaults to None.

**Attributes**

| | |
|---|---|
| **message_type** | None |

**Methods**

| | |
|---|---|
| *to_dict* | Return a dictionary form of the data check message. |

**to_dict**(*self*)

Return a dictionary form of the data check message.

**class** evalml.data_checks.data_check_message.**DataCheckWarning**(*message*, *data_check_name*, *message_code=None*, *details=None*, *action_options=None*)

DataCheckMessage subclass for warnings returned by data checks.

**Attributes**

| | |
|---|---|
| **message_type** | DataCheckMessageType.WARNING |

**Methods**

| | |
|---|---|
| *to_dict* | Return a dictionary form of the data check message. |

`to_dict`(*self*)

Return a dictionary form of the data check message.

## data_check_message_code

Enum for data check message code.

### Module Contents

### Classes Summary

| | |
|---|---|
| *DataCheckMessageCode* | Enum for data check message code. |

### Contents

`class` `evalml.data_checks.data_check_message_code.`**`DataCheckMessageCode`**

Enum for data check message code.

**Attributes**

| | |
|---|---|
| **CLASS_IMBALANCE_BELOW_FOLDS** | Message code for when the number of values for each target is below 2 * number of CV folds. |
| **CLASS_IMBALANCE_BELOW_THRESHOLD** | Message code for when balance in classes is less than the threshold. |
| **CLASS_IMBALANCE_SEVERE** | Message code for when balance in classes is less than the threshold and minimum class is less than minimum number of accepted samples. |
| **COLS_WITH_NULL** | Message code for columns with null values. |
| **DATETIME_HAS_MISALIGNED_VALUES** | Message code for when datetime information has values that are not aligned with the inferred frequency. |
| **DATETIME_HAS_NAN** | Message code for when input datetime columns contain NaN values. |
| **DATETIME_HAS_REDUNDANT_ROW** | Message code for when datetime information has more than one row per datetime. |
| **DATETIME_HAS_UNEVEN_INTERVALS** | Message code for when the datetime values have uneven intervals. |
| **DATETIME_INFORMATION_NOT_FOUND** | Message code for when datetime information can not be found or is in an unaccepted format. |

continues on next page

Table 1 – continued from previous page

| | |
|---|---|
| **DATETIME_IS_MISSING_VALUES** | Message code for when datetime feature has values missing between the start and end dates. |
| **DATETIME_IS_NOT_MONOTONIC** | Message code for when the datetime values are not monotonically increasing. |
| **DATETIME_NO_FREQUENCY_INFERRED** | Message code for when no frequency can be inferred in the datetime values through Woodwork infer_frequency. |
| **HAS_ID_COLUMN** | Message code for data that has ID columns. |
| **HAS_ID_FIRST_COLUMN** | Message code for data that has an ID column as the first column. |
| **HAS_OUTLIERS** | Message code for when outliers are detected. |
| **HIGH_VARIANCE** | Message code for when high variance is detected for cross-validation. |
| **HIGHLY_NULL_COLS** | Message code for highly null columns. |
| **HIGHLY_NULL_ROWS** | Message code for highly null rows. |
| **IS_MULTICOLLINEAR** | Message code for when data is potentially multicollinear. |
| **MISMATCHED_INDICES** | Message code for when input target and features have mismatched indices. |
| **MISMATCHED_INDICES_ORDER** | Message code for when input target and features have mismatched indices order. The two inputs have the same index values, but shuffled. |
| **MISMATCHED_LENGTHS** | Message code for when input target and features have different lengths. |
| **NATURAL_LANGUAGE_HAS_NAN** | Message code for when input natural language columns contain NaN values. |
| **NO_VARIANCE** | Message code for when data has no variance (1 unique value). |
| **NO_VARIANCE_WITH_NULL** | Message code for when data has one unique value and NaN values. |
| **NO_VARIANCE_ZERO_UNIQUE** | Message code for when data has no variance (0 unique value) |
| **NOT_UNIQUE_ENOUGH** | Message code for when data does not possess enough unique values. |
| **TARGET_BINARY_NOT_TWO_UNIQUE_VALUES** | Message code for target data for a binary classification problem that does not have two unique values. |
| **TARGET_HAS_NULL** | Message code for target data that has null values. |
| **TARGET_INCOMPATIBLE_OBJECTIVE** | Message code for target data that has incompatible values for the specified objective |
| **TARGET_IS_EMPTY_OR_FULLY_NULL** | Message code for target data that is empty or has all null values. |
| **TARGET_IS_NONE** | Message code for when target is None. |
| **TARGET_LEAKAGE** | Message code for when target leakage is detected. |
| **TARGET_LOGNORMAL_DISTRIBUTION** | Message code for target data with a lognormal distribution. |
| **TARGET_MULTICLASS_HIGH_UNIQUE_CLASS** | Message code for target data for a multi classification problem that has an abnormally large number of unique classes relative to the number of target values. |
| **TARGET_MULTICLASS_NOT_ENOUGH_CLASSES** | Message code for target data for a multi classification problem that does not have more than two unique classes. |
| **TARGET_MULTICLASS_NOT_TWO_EXAMPLES_PER_CLASS** | Message code for target data for a multi classification problem that does not have two examples per class. |
| **TARGET_UNSUPPORTED_PROBLEM_TYPE** | Message code for target data that is being checked against an unsupported problem type. |
| **TARGET_UNSUPPORTED_TYPE** | Message code for target data that is of an unsupported type. |
| **TARGET_UNSUPPORTED_TYPE_REGRESSION** | Message code for target data that is incompatible with regression |

continues on next page

Table 1 – continued from previous page

| | |
|---|---|
| **TIME-SERIES_PARAMETERS_NOT_COMPATIBLE_WITH_SPLIT** | Message code when the time series parameters are too large for the smallest data split. |
| **TIME-SERIES_TARGET_NOT_COMPATIBLE_WITH_SPLIT** | Message code when any training and validation split of the time series target doesn't contain all classes. |
| **TOO_SPARSE** | Message code for when multiclass data has values that are too sparsely populated. |
| **TOO_UNIQUE** | Message code for when data possesses too many unique values. |

**Methods**

| | |
|---|---|
| *name* | The name of the Enum member. |
| *value* | The value of the Enum member. |

**name**(*self*)

The name of the Enum member.

**value**(*self*)

The value of the Enum member.

## data_check_message_type

Enum for type of data check message.

### Module Contents

### Classes Summary

| | |
|---|---|
| *DataCheckMessageType* | Enum for type of data check message: WARNING or ERROR. |

### Contents

**class** evalml.data_checks.data_check_message_type.**DataCheckMessageType**

Enum for type of data check message: WARNING or ERROR.

**Attributes**

| | |
|---|---|
| **ERROR** | Error message returned by a data check. |
| **WARNING** | Warning message returned by a data check. |

**Methods**

| | |
|---|---|
| *name* | The name of the Enum member. |
| *value* | The value of the Enum member. |

**name**(*self*)

The name of the Enum member.

**value**(*self*)

The value of the Enum member.

### data_checks

A collection of data checks.

### Module Contents

### Classes Summary

| | |
|---|---|
| *DataChecks* | A collection of data checks. |

### Contents

**class** evalml.data_checks.data_checks.**DataChecks**(*data_checks=None*, *data_check_params=None*)

A collection of data checks.

**Parameters**

- **data_checks** (*list (DataCheck)*) – List of DataCheck objects.
- **data_check_params** (*dict*) – Parameters for passed DataCheck objects.

**Methods**

| | |
|---|---|
| *validate* | Inspect and validate the input data against data checks and returns a list of warnings and errors if applicable. |

**validate**(*self*, *X*, *y=None*)

Inspect and validate the input data against data checks and returns a list of warnings and errors if applicable.

**Parameters**

- **X** (*pd.DataFrame, np.ndarray*) – The input data of shape [n_samples, n_features]
- **y** (*pd.Series, np.ndarray*) – The target data of length [n_samples]

**Returns** Dictionary containing DataCheckMessage objects

**Return type** dict

### datetime_format_data_check

Data check that checks if the datetime column has equally spaced intervals and is monotonically increasing or decreasing in order to be supported by time series estimators.

## Module Contents

## Classes Summary

| | |
|---|---|
| *DateTimeFormatDataCheck* | Check if the datetime column has equally spaced intervals and is monotonically increasing or decreasing in order to be supported by time series estimators. |

## Contents

**class** evalml.data_checks.datetime_format_data_check.**DateTimeFormatDataCheck**(*datetime_column='index', nan_duplicate_threshold=0.75*)

Check if the datetime column has equally spaced intervals and is monotonically increasing or decreasing in order to be supported by time series estimators.

**Parameters**

- **datetime_column** (*str, int*) – The name of the datetime column. If the datetime values are in the index, then pass "index".
- **nan_duplicate_threshold** (*float*) – The percentage of values in the *datetime_column* that must not be duplicate or nan before *DATETIME_NO_FREQUENCY_INFERRED* is returned instead of *DATETIME_HAS_UNEVEN_INTERVALS*. For example, if this is set to 0.80, then only 20% of the values in *datetime_column* can be duplicate or nan. Defaults to 0.75.

**Methods**

| | |
|---|---|
| *name* | Return a name describing the data check. |
| *validate* | Checks if the target data has equal intervals and is monotonically increasing. |

**name**(*cls*)

Return a name describing the data check.

**validate**(*self, X, y*)

Checks if the target data has equal intervals and is monotonically increasing.

Will return a DataCheckError if the data is not a datetime type, is not increasing, has redundant or missing row(s), contains invalid (NaN or None) values, or has values that don't align with the assumed frequency.

**Parameters**

- **X** (*pd.DataFrame, np.ndarray*) – Features.
- **y** (*pd.Series, np.ndarray*) – Target data.

**Returns** List with DataCheckErrors if unequal intervals are found in the datetime column.

**Return type** dict (DataCheckError)

### Examples

```
>>> import pandas as pd
```

The column 'dates' has a set of two dates with daily frequency, two dates with hourly frequency, and two dates with monthly frequency.

```
>>> X = pd.DataFrame(pd.date_range("2015-01-01", periods=2).append(pd.date_
↪range("2015-01-08", periods=2, freq="H").append(pd.date_range("2016-03-02",␣
↪periods=2, freq="M"))), columns=["dates"])
>>> y = pd.Series([0, 1, 0, 1, 1, 0])
>>> datetime_format_dc = DateTimeFormatDataCheck(datetime_column="dates")
>>> assert datetime_format_dc.validate(X, y) == [
...     {
...         "message": "No frequency could be detected in column 'dates',␣
↪possibly due to uneven intervals or too many duplicate/missing values.",
...         "data_check_name": "DateTimeFormatDataCheck",
...         "level": "error",
...         "code": "DATETIME_NO_FREQUENCY_INFERRED",
...         "details": {"columns": None, "rows": None},
...         "action_options": []
...      }
... ]
```

The column "dates" has a gap in the values, which implies there are many dates missing.

```
>>> X = pd.DataFrame(pd.date_range("2021-01-01", periods=9).append(pd.date_
↪range("2021-01-31", periods=50)), columns=["dates"])
>>> y = pd.Series([0, 1, 0, 1, 1, 0, 0, 0, 1, 0])
>>> ww_payload = infer_frequency(X["dates"], debug=True, window_length=5,␣
↪threshold=0.8)
>>> datetime_format_dc = DateTimeFormatDataCheck(datetime_column="dates")
>>> assert datetime_format_dc.validate(X, y) == [
...     {
...         "message": "Column 'dates' has datetime values missing between␣
↪start and end date.",
...         "data_check_name": "DateTimeFormatDataCheck",
...         "level": "error",
...         "code": "DATETIME_IS_MISSING_VALUES",
...         "details": {"columns": None, "rows": None},
...         "action_options": []
...      },
...     {
...         "message": "A frequency was detected in column 'dates', but there␣
↪are faulty datetime values that need to be addressed.",
...         "data_check_name": "DateTimeFormatDataCheck",
...         "level": "error",
...         "code": "DATETIME_HAS_UNEVEN_INTERVALS",
...         "details": {'columns': None, 'rows': None},
...         "action_options": [
...             {
...                 'code': 'REGULARIZE_AND_IMPUTE_DATASET',
...                 'data_check_name': 'DateTimeFormatDataCheck',
```

(continues on next page)

(continued from previous page)

```
...                 'metadata': {
...                         'columns': None,
...                         'is_target': True,
...                         'rows': None
...                 },
...                 'parameters': {
...                         'time_index': {
...                             'default_value': 'dates',
...                             'parameter_type': 'global',
...                             'type': 'str'
...                         },
...                         'frequency_payload': {
...                             'default_value': ww_payload,
...                             'parameter_type': 'global',
...                             'type': 'tuple'
...                         }
...                 }
...             }
...         ]
...     }
... ]
```

The column "dates" has a repeat of the date 2021-01-09 appended to the end, which is considered redundant and will raise an error.

```
>>> X = pd.DataFrame(pd.date_range("2021-01-01", periods=9).append(pd.date_
↪range("2021-01-09", periods=1)), columns=["dates"])
>>> y = pd.Series([0, 1, 0, 1, 1, 0, 0, 0, 1, 0])
>>> ww_payload = infer_frequency(X["dates"], debug=True, window_length=5,␣
↪threshold=0.8)
>>> datetime_format_dc = DateTimeFormatDataCheck(datetime_column="dates")
>>> assert datetime_format_dc.validate(X, y) == [
...     {
...         "message": "Column 'dates' has more than one row with the same␣
↪datetime value.",
...         "data_check_name": "DateTimeFormatDataCheck",
...         "level": "error",
...         "code": "DATETIME_HAS_REDUNDANT_ROW",
...         "details": {"columns": None, "rows": None},
...         "action_options": []
...      },
...     {
...         "message": "A frequency was detected in column 'dates', but there␣
↪are faulty datetime values that need to be addressed.",
...         "data_check_name": "DateTimeFormatDataCheck",
...         "level": "error",
...         "code": "DATETIME_HAS_UNEVEN_INTERVALS",
...         "details": {'columns': None, 'rows': None},
...         "action_options": [
...             {
...                 'code': 'REGULARIZE_AND_IMPUTE_DATASET',
...                 'data_check_name': 'DateTimeFormatDataCheck',
```

(continues on next page)

(continued from previous page)

```
...                 'metadata': {
...                         'columns': None,
...                         'is_target': True,
...                         'rows': None
...                 },
...                 'parameters': {
...                         'time_index': {
...                             'default_value': 'dates',
...                             'parameter_type': 'global',
...                             'type': 'str'
...                         },
...                         'frequency_payload': {
...                             'default_value': ww_payload,
...                             'parameter_type': 'global',
...                             'type': 'tuple'
...                         }
...                 }
...             }
...         ]
...     }
... ]
```

The column "Weeks" has a date that does not follow the weekly pattern, which is considered misaligned.

```
>>> X = pd.DataFrame(pd.date_range("2021-01-01", freq="W", periods=12).
↪append(pd.date_range("2021-03-22", periods=1)), columns=["Weeks"])
>>> ww_payload = infer_frequency(X["Weeks"], debug=True, window_length=5,␣
↪threshold=0.8)
>>> datetime_format_dc = DateTimeFormatDataCheck(datetime_column="Weeks")
>>> assert datetime_format_dc.validate(X, y) == [
...     {
...         "message": "Column 'Weeks' has datetime values that do not align␣
↪with the inferred frequency.",
...         "data_check_name": "DateTimeFormatDataCheck",
...         "level": "error",
...         "details": {"columns": None, "rows": None},
...         "code": "DATETIME_HAS_MISALIGNED_VALUES",
...         "action_options": []
...      },
...     {
...         "message": "A frequency was detected in column 'Weeks', but there␣
↪are faulty datetime values that need to be addressed.",
...         "data_check_name": "DateTimeFormatDataCheck",
...         "level": "error",
...         "code": "DATETIME_HAS_UNEVEN_INTERVALS",
...         "details": {'columns': None, 'rows': None},
...         "action_options": [
...             {
...                 'code': 'REGULARIZE_AND_IMPUTE_DATASET',
...                 'data_check_name': 'DateTimeFormatDataCheck',
...                 'metadata': {
...                         'columns': None,
```

(continues on next page)

(continued from previous page)

```
...                         'is_target': True,
...                         'rows': None
...                 },
...                 'parameters': {
...                         'time_index': {
...                             'default_value': 'Weeks',
...                             'parameter_type': 'global',
...                             'type': 'str'
...                         },
...                         'frequency_payload': {
...                             'default_value': ww_payload,
...                             'parameter_type': 'global',
...                             'type': 'tuple'
...                         }
...                 }
...             }
...         ]
...     }
... ]
```

The column "Weeks" has a date that does not follow the weekly pattern, which is considered misaligned.

```
>>> X = pd.DataFrame(pd.date_range("2021-01-01", freq="W", periods=12).
↪append(pd.date_range("2021-03-22", periods=1)), columns=["Weeks"])
>>> ww_payload = infer_frequency(X["Weeks"], debug=True, window_length=5,␣
↪threshold=0.8)
>>> datetime_format_dc = DateTimeFormatDataCheck(datetime_column="Weeks")
>>> assert datetime_format_dc.validate(X, y) == [
...     {
...         "message": "Column 'Weeks' has datetime values that do not align␣
↪with the inferred frequency.",
...         "data_check_name": "DateTimeFormatDataCheck",
...         "level": "error",
...         "details": {"columns": None, "rows": None},
...         "code": "DATETIME_HAS_MISALIGNED_VALUES",
...         "action_options": []
...      },
...     {
...         "message": "A frequency was detected in column 'Weeks', but there␣
↪are faulty datetime values that need to be addressed.",
...         "data_check_name": "DateTimeFormatDataCheck",
...         "level": "error",
...         "code": "DATETIME_HAS_UNEVEN_INTERVALS",
...         "details": {'columns': None, 'rows': None},
...         "action_options": [
...             {
...                 'code': 'REGULARIZE_AND_IMPUTE_DATASET',
...                 'data_check_name': 'DateTimeFormatDataCheck',
...                 'metadata': {
...                         'columns': None,
...                         'is_target': True,
...                         'rows': None
```

(continues on next page)

(continued from previous page)

```
...                 },
...                 'parameters': {
...                         'time_index': {
...                             'default_value': 'Weeks',
...                             'parameter_type': 'global',
...                             'type': 'str'
...                         },
...                         'frequency_payload': {
...                             'default_value': ww_payload,
...                             'parameter_type': 'global',
...                             'type': 'tuple'
...                         }
...                 }
...             }
...         ]
...     }
... ]
```

The column "Weeks" passed integers instead of datetime data, which will raise an error.

```
>>> X = pd.DataFrame([1, 2, 3, 4], columns=["Weeks"])
>>> y = pd.Series([0] * 4)
>>> datetime_format_dc = DateTimeFormatDataCheck(datetime_column="Weeks")
>>> assert datetime_format_dc.validate(X, y) == [
...     {
...         "message": "Datetime information could not be found in the data, or␣
↪was not in a supported datetime format.",
...         "data_check_name": "DateTimeFormatDataCheck",
...         "level": "error",
...         "details": {"columns": None, "rows": None},
...         "code": "DATETIME_INFORMATION_NOT_FOUND",
...         "action_options": []
...      }
... ]
```

Converting that same integer data to datetime, however, is valid.

```
>>> X = pd.DataFrame(pd.to_datetime([1, 2, 3, 4]), columns=["Weeks"])
>>> datetime_format_dc = DateTimeFormatDataCheck(datetime_column="Weeks")
>>> assert datetime_format_dc.validate(X, y) == []
```

```
>>> X = pd.DataFrame(pd.date_range("2021-01-01", freq="W", periods=10),␣
↪columns=["Weeks"])
>>> datetime_format_dc = DateTimeFormatDataCheck(datetime_column="Weeks")
>>> assert datetime_format_dc.validate(X, y) == []
```

While the data passed in is of datetime type, time series requires the datetime information in datetime_column to be monotonically increasing (ascending).

```
>>> X = X.iloc[::-1]
>>> datetime_format_dc = DateTimeFormatDataCheck(datetime_column="Weeks")
>>> assert datetime_format_dc.validate(X, y) == [
```

(continues on next page)

(continued from previous page)

```
...     {
...         "message": "Datetime values must be sorted in ascending order.",
...         "data_check_name": "DateTimeFormatDataCheck",
...         "level": "error",
...         "details": {"columns": None, "rows": None},
...         "code": "DATETIME_IS_NOT_MONOTONIC",
...         "action_options": []
...      }
... ]
```

The first value in the column "index" is replaced with NaT, which will raise an error in this data check.

```
>>> dates = [["2-1-21", "3-1-21"],
...         ["2-2-21", "3-2-21"],
...         ["2-3-21", "3-3-21"],
...         ["2-4-21", "3-4-21"],
...         ["2-5-21", "3-5-21"],
...         ["2-6-21", "3-6-21"],
...         ["2-7-21", "3-7-21"],
...         ["2-8-21", "3-8-21"],
...         ["2-9-21", "3-9-21"],
...         ["2-10-21", "3-10-21"],
...         ["2-11-21", "3-11-21"],
...         ["2-12-21", "3-12-21"]]
>>> dates[0][0] = None
>>> df = pd.DataFrame(dates, columns=["days", "days2"])
>>> ww_payload = infer_frequency(pd.to_datetime(df["days"]), debug=True, window_
↪length=5, threshold=0.8)
>>> datetime_format_dc = DateTimeFormatDataCheck(datetime_column="days")
>>> assert datetime_format_dc.validate(df, y) == [
...     {
...         "message": "Input datetime column 'days' contains NaN values.␣
↪Please impute NaN values or drop these rows.",
...         "data_check_name": "DateTimeFormatDataCheck",
...         "level": "error",
...         "details": {"columns": None, "rows": None},
...         "code": "DATETIME_HAS_NAN",
...         "action_options": []
...      },
...     {
...         "message": "A frequency was detected in column 'days', but there␣
↪are faulty datetime values that need to be addressed.",
...         "data_check_name": "DateTimeFormatDataCheck",
...         "level": "error",
...         "code": "DATETIME_HAS_UNEVEN_INTERVALS",
...         "details": {'columns': None, 'rows': None},
...         "action_options": [
...             {
...                 'code': 'REGULARIZE_AND_IMPUTE_DATASET',
...                 'data_check_name': 'DateTimeFormatDataCheck',
...                 'metadata': {
...                         'columns': None,
```

(continues on next page)

(continued from previous page)

```
...                         'is_target': True,
...                         'rows': None
...                 },
...                 'parameters': {
...                         'time_index': {
...                             'default_value': 'days',
...                             'parameter_type': 'global',
...                             'type': 'str'
...                         },
...                         'frequency_payload': {
...                             'default_value': ww_payload,
...                             'parameter_type': 'global',
...                             'type': 'tuple'
...                         }
...                 }
...             }
...         ]
...     }
... ]
...
```

### default_data_checks

A default set of data checks that can be used for a variety of datasets.

### Module Contents

### Classes Summary

| | |
|---|---|
| *DefaultDataChecks* | A collection of basic data checks that is used by AutoML by default. |

### Contents

**class** evalml.data_checks.default_data_checks.**DefaultDataChecks**(*problem_type*, *objective*, *n_splits=3*, *problem_configuration=None*)

A collection of basic data checks that is used by AutoML by default.

Includes:

- *NullDataCheck*
- *HighlyNullRowsDataCheck*
- *IDColumnsDataCheck*
- *TargetLeakageDataCheck*
- *InvalidTargetDataCheck*

- *NoVarianceDataCheck*
- *ClassImbalanceDataCheck* (for classification problem types)
- *TargetDistributionDataCheck* (for regression problem types)
- *DateTimeFormatDataCheck* (for time series problem types)
- 'TimeSeriesParametersDataCheck' (for time series problem types)
- *TimeSeriesSplittingDataCheck* (for time series classification problem types)

**Parameters**

- **problem_type** (*str*) – The problem type that is being validated. Can be regression, binary, or multiclass.
- **objective** (*str or ObjectiveBase*) – Name or instance of the objective class.
- **n_splits** (*int*) – The number of splits as determined by the data splitter being used. Defaults to 3.
- **problem_configuration** (*dict*) – Required for time series problem types. Values should be passed in for time_index,
- **gap** –
- **forecast_horizon** –
- **max_delay.** (*and*) –

**Methods**

| | |
|---|---|
| *validate* | Inspect and validate the input data against data checks and returns a list of warnings and errors if applicable. |

**validate**(*self*, *X*, *y=None*)

Inspect and validate the input data against data checks and returns a list of warnings and errors if applicable.

**Parameters**

- **X** (*pd.DataFrame, np.ndarray*) – The input data of shape [n_samples, n_features]
- **y** (*pd.Series, np.ndarray*) – The target data of length [n_samples]

**Returns** Dictionary containing DataCheckMessage objects

**Return type** dict

### id_columns_data_check

Data check that checks if any of the features are likely to be ID columns.

### Module Contents

### Classes Summary

| | |
|---|---|
| *IDColumnsDataCheck* | Check if any of the features are likely to be ID columns. |

### Contents

**class** evalml.data_checks.id_columns_data_check.**IDColumnsDataCheck**(*id_threshold=1.0*)

Check if any of the features are likely to be ID columns.

**Parameters** **id_threshold** (*float*) – The probability threshold to be considered an ID column. Defaults to 1.0.

**Methods**

| | |
|---|---|
| *name* | Return a name describing the data check. |
| *validate* | Check if any of the features are likely to be ID columns. Currently performs a number of simple checks. |

**name**(*cls*)

Return a name describing the data check.

**validate**(*self*, *X*, *y=None*)

Check if any of the features are likely to be ID columns. Currently performs a number of simple checks.

Checks performed are:

- column name is "id"
- column name ends in "_id"
- column contains all unique values (and is categorical / integer type)

**Parameters**

- **X** (*pd.DataFrame, np.ndarray*) – The input features to check.
- **y** (*pd.Series*) – The target. Defaults to None. Ignored.

**Returns** A dictionary of features with column name or index and their probability of being ID columns

**Return type** dict

### Examples

```
>>> import pandas as pd
```

Columns that end in "_id" and are completely unique are likely to be ID columns.

```
>>> df = pd.DataFrame({
...     "profits": [25, 15, 15, 31, 19],
...     "customer_id": [123, 124, 125, 126, 127],
...     "Sales": [10, 42, 31, 51, 61]
... })
...
>>> id_col_check = IDColumnsDataCheck()
>>> assert id_col_check.validate(df) == [
...     {
...         "message": "Columns 'customer_id' are 100.0% or more likely to be␣
↪an ID column",
...         "data_check_name": "IDColumnsDataCheck",
...         "level": "warning",
...         "code": "HAS_ID_COLUMN",
...         "details": {"columns": ["customer_id"], "rows": None},
...         "action_options": [
...             {
...                 "code": "DROP_COL",
...                 "data_check_name": "IDColumnsDataCheck",
...                 "parameters": {},
...                 "metadata": {"columns": ["customer_id"], "rows": None}
...             }
...         ]
...     }
... ]
```

Columns named "ID" with all unique values will also be identified as ID columns.

```
>>> df = df.rename(columns={"customer_id": "ID"})
>>> id_col_check = IDColumnsDataCheck()
>>> assert id_col_check.validate(df) == [
...     {
...         "message": "Columns 'ID' are 100.0% or more likely to be an ID␣
↪column",
...         "data_check_name": "IDColumnsDataCheck",
...         "level": "warning",
...         "code": "HAS_ID_COLUMN",
...         "details": {"columns": ["ID"], "rows": None},
...         "action_options": [
...            {
...                 "code": "DROP_COL",
...                 "data_check_name": "IDColumnsDataCheck",
...                 "parameters": {},
...                 "metadata": {"columns": ["ID"], "rows": None}
...             }
...         ]
...     }
... ]
```

Despite being all unique, "Country_Rank" will not be identified as an ID column as id_threshold is set to 1.0 by default and its name doesn't indicate that it's an ID.

```
>>> df = pd.DataFrame({
...     "humidity": ["high", "very high", "low", "low", "high"],
...     "Country_Rank": [1, 2, 3, 4, 5],
...     "Sales": ["very high", "high", "high", "medium", "very low"]
... })
...
>>> id_col_check = IDColumnsDataCheck()
>>> assert id_col_check.validate(df) == []
```

However lowering the threshold will cause this column to be identified as an ID.

```
>>> id_col_check = IDColumnsDataCheck()
>>> id_col_check = IDColumnsDataCheck(id_threshold=0.95)
>>> assert id_col_check.validate(df) == [
...     {
...         "message": "Columns 'Country_Rank' are 95.0% or more likely to be␣
↪an ID column",
...         "data_check_name": "IDColumnsDataCheck",
...         "level": "warning",
...         "details": {"columns": ["Country_Rank"], "rows": None},
...         "code": "HAS_ID_COLUMN",
...         "action_options": [
...             {
...                 "code": "DROP_COL",
...                 "data_check_name": "IDColumnsDataCheck",
...                 "parameters": {},
...                 "metadata": {"columns": ["Country_Rank"], "rows": None}
...             }
...         ]
...     }
... ]
```

If the first column of the dataframe has all unique values and is named either 'ID' or a name that ends with '_id', it is probably the primary key. The other ID columns should be dropped.

```
>>> df = pd.DataFrame({
...     "sales_id": [0, 1, 2, 3, 4],
...     "customer_id": [123, 124, 125, 126, 127],
...     "Sales": [10, 42, 31, 51, 61]
... })
...
>>> id_col_check = IDColumnsDataCheck()
>>> assert id_col_check.validate(df) == [
...     {
...         "message": "The first column 'sales_id' is likely to be the primary␣
↪key",
...         "data_check_name": "IDColumnsDataCheck",
...         "level": "warning",
...         "code": "HAS_ID_FIRST_COLUMN",
...         "details": {"columns": ["sales_id"], "rows": None},
...         "action_options": [
```

(continues on next page)

(continued from previous page)

```
...             {
...                 "code": "SET_FIRST_COL_ID",
...                 "data_check_name": "IDColumnsDataCheck",
...                 "parameters": {},
...                 "metadata": {"columns": ["sales_id"], "rows": None}
...             }
...         ]
...     },
...     {
...         "message": "Columns 'customer_id' are 100.0% or more likely to be an␣
↪ID column",
...         "data_check_name": "IDColumnsDataCheck",
...         "level": "warning",
...         "code": "HAS_ID_COLUMN",
...         "details": {"columns": ["customer_id"], "rows": None},
...         "action_options": [
...             {
...                 "code": "DROP_COL",
...                 "data_check_name": "IDColumnsDataCheck",
...                 "parameters": {},
...                 "metadata": {"columns": ["customer_id"], "rows": None}
...             }
...         ]
...     }
... ]
```

### invalid_target_data_check

Data check that checks if the target data contains missing or invalid values.

### Module Contents

### Classes Summary

| | |
|---|---|
| *InvalidTargetDataCheck* | Check if the target data is considered invalid. |

### Contents

**class** evalml.data_checks.invalid_target_data_check.**InvalidTargetDataCheck**(*problem_type*, *objective*, *n_unique=100*, *null_strategy='drop'*)

Check if the target data is considered invalid.

**Target data is considered invalid if:**

- Target is None.
- Target has NaN or None values.

- Target is of an unsupported Woodwork logical type.
- Target and features have different lengths or indices.
- Target does not have enough instances of a class in a classification problem.
- Target does not contain numeric data for regression problems.

**Parameters**

- **problem_type** (*str or ProblemTypes*) – The specific problem type to data check for. e.g. 'binary', 'multiclass', 'regression, 'time series regression'
- **objective** (*str or ObjectiveBase*) – Name or instance of the objective class.
- **n_unique** (*int*) – Number of unique target values to store when problem type is binary and target incorrectly has more than 2 unique values. Non-negative integer. If None, stores all unique values. Defaults to 100.
- **null_strategy** (*str*) – The type of action option that should be returned if the target is partially null. The options are *impute* and *drop* (default). *impute* - Will return a *DataCheckActionOption* for imputing the target column. *drop* - Will return a *DataCheckActionOption* for dropping the null rows in the target column.

**Attributes**

| | |
|---|---|
| **multi-class_continuous_threshold** | 0.05 |

**Methods**

| | |
|---|---|
| *name* | Return a name describing the data check. |
| *validate* | Check if the target data is considered invalid. If the input features argument is not None, it will be used to check that the target and features have the same dimensions and indices. |

**name**(*cls*)

Return a name describing the data check.

**validate**(*self*, *X*, *y*)

Check if the target data is considered invalid. If the input features argument is not None, it will be used to check that the target and features have the same dimensions and indices.

**Target data is considered invalid if:**

- Target is None.
- Target has NaN or None values.
- Target is of an unsupported Woodwork logical type.
- Target and features have different lengths or indices.
- Target does not have enough instances of a class in a classification problem.
- Target does not contain numeric data for regression problems.

**Parameters**

- **X** (*pd.DataFrame, np.ndarray*) – Features. If not None, will be used to check that the target and features have the same dimensions and indices.
- **y** (*pd.Series, np.ndarray*) – Target data to check for invalid values.

**Returns** List with DataCheckErrors if any invalid values are found in the target data.

**Return type** dict (DataCheckError)

### Examples

```
>>> import pandas as pd
```

Target values must be integers, doubles, or booleans.

```
>>> X = pd.DataFrame({"col": [1, 2, 3, 1]})
>>> y = pd.Series(["cat_1", "cat_2", "cat_1", "cat_2"])
>>> target_check = InvalidTargetDataCheck("regression", "R2", null_strategy=
↪"impute")
>>> assert target_check.validate(X, y) == [
...     {
...         "message": "Target is unsupported Unknown type. Valid Woodwork␣
↪logical types include: integer, double, boolean, age, age_fractional, integer_
↪nullable, boolean_nullable, age_nullable",
...         "data_check_name": "InvalidTargetDataCheck",
...         "level": "error",
...         "details": {"columns": None, "rows": None, "unsupported_type":
↪"unknown"},
...         "code": "TARGET_UNSUPPORTED_TYPE",
...         "action_options": []
...     },
...     {
...         "message": "Target data type should be numeric for regression type␣
↪problems.",
...         "data_check_name": "InvalidTargetDataCheck",
...         "level": "error",
...         "details": {"columns": None, "rows": None},
...         "code": "TARGET_UNSUPPORTED_TYPE_REGRESSION",
...         "action_options": []
...     }
... ]
```

The target cannot have null values.

```
>>> y = pd.Series([None, pd.NA, pd.NaT, None])
>>> assert target_check.validate(X, y) == [
...     {
...         "message": "Target is either empty or fully null.",
...         "data_check_name": "InvalidTargetDataCheck",
...         "level": "error",
...         "details": {"columns": None, "rows": None},
...         "code": "TARGET_IS_EMPTY_OR_FULLY_NULL",
...         "action_options": []
...     }
```

(continues on next page)

(continued from previous page)

```
... ]
...
...
>>> y = pd.Series([1, None, 3, None])
>>> assert target_check.validate(None, y) == [
...     {
...         "message": "2 row(s) (50.0%) of target values are null",
...         "data_check_name": "InvalidTargetDataCheck",
...         "level": "error",
...         "details": {
...             "columns": None,
...             "rows": [1, 3],
...             "num_null_rows": 2,
...             "pct_null_rows": 50.0
...         },
...         "code": "TARGET_HAS_NULL",
...         "action_options": [
...            {
...                 "code": "IMPUTE_COL",
...                 "data_check_name": "InvalidTargetDataCheck",
...                 "parameters": {
...                     "impute_strategy": {
...                         "parameter_type": "global",
...                         "type": "category",
...                         "categories": ["mean", "most_frequent"],
...                         "default_value": "mean"
...                     }
...                 },
...                 "metadata": {"columns": None, "rows": None, "is_target":␣
↪True},
...              }
...         ],
...     }
... ]
```

If the target values don't match the problem type passed, an error will be raised. In this instance, only two values exist in the target column, but multiclass has been passed as the problem type.

```
>>> X = pd.DataFrame([i for i in range(50)])
>>> y = pd.Series([i%2 for i in range(50)])
>>> target_check = InvalidTargetDataCheck("multiclass", "Log Loss Multiclass")
>>> assert target_check.validate(X, y) == [
...     {
...         "message": "Target has two or less classes, which is too few for␣
↪multiclass problems.  Consider changing to binary.",
...         "data_check_name": "InvalidTargetDataCheck",
...         "level": "error",
...         "details": {"columns": None, "rows": None, "num_classes": 2},
...         "code": "TARGET_MULTICLASS_NOT_ENOUGH_CLASSES",
...         "action_options": []
...     }
... ]
```

If the length of X and y differ, a warning will be raised. A warning will also be raised for indices that don"t match.

```
>>> target_check = InvalidTargetDataCheck("regression", "R2")
>>> X = pd.DataFrame([i for i in range(5)])
>>> y = pd.Series([1, 2, 4, 3], index=[1, 2, 4, 3])
>>> assert target_check.validate(X, y) == [
...     {
...         "message": "Input target and features have different lengths",
...         "data_check_name": "InvalidTargetDataCheck",
...         "level": "warning",
...         "details": {"columns": None, "rows": None, "features_length": 5,
↪"target_length": 4},
...         "code": "MISMATCHED_LENGTHS",
...         "action_options": []
...     },
...     {
...         "message": "Input target and features have mismatched indices.␣
↪Details will include the first 10 mismatched indices.",
...         "data_check_name": "InvalidTargetDataCheck",
...         "level": "warning",
...         "details": {
...             "columns": None,
...             "rows": None,
...             "indices_not_in_features": [],
...             "indices_not_in_target": [0]
...         },
...         "code": "MISMATCHED_INDICES",
...         "action_options": []
...     }
... ]
```

### multicollinearity_data_check

Data check to check if any set features are likely to be multicollinear.

### Module Contents

### Classes Summary

| | |
|---|---|
| *MulticollinearityDataCheck* | Check if any set features are likely to be multicollinear. |

### Contents

**class** evalml.data_checks.multicollinearity_data_check.**MulticollinearityDataCheck**(*threshold=0.9*)

Check if any set features are likely to be multicollinear.

**Parameters** **threshold** (*float*) – The threshold to be considered. Defaults to 0.9.

**Methods**

| | |
|---|---|
| *name* | Return a name describing the data check. |
| *validate* | Check if any set of features are likely to be multicollinear. |

**name**(*cls*)

Return a name describing the data check.

**validate**(*self*, *X*, *y=None*)

Check if any set of features are likely to be multicollinear.

**Parameters**

- **X** (*pd.DataFrame*) – The input features to check.
- **y** (*pd.Series*) – The target. Ignored.

**Returns** dict with a DataCheckWarning if there are any potentially multicollinear columns.

**Return type** dict

#### Example

```
>>> import pandas as pd
```

Columns in X that are highly correlated with each other will be identified using mutual information.

```
>>> col = pd.Series([1, 0, 2, 3, 4] * 15)
>>> X = pd.DataFrame({"col_1": col, "col_2": col * 3})
>>> y = pd.Series([1, 0, 0, 1, 0] * 15)
...
>>> multicollinearity_check = MulticollinearityDataCheck(threshold=1.0)
>>> assert multicollinearity_check.validate(X, y) == [
...     {
...         "message": "Columns are likely to be correlated: [('col_1', 'col_2
↪')]",
...         "data_check_name": "MulticollinearityDataCheck",
...         "level": "warning",
...         "code": "IS_MULTICOLLINEAR",
...         "details": {"columns": [("col_1", "col_2")], "rows": None},
...         "action_options": []
...     }
... ]
```

### no_variance_data_check

Data check that checks if the target or any of the features have no variance.

### Module Contents

### Classes Summary

| | |
|---|---|
| *NoVarianceDataCheck* | Check if the target or any of the features have no variance. |

### Contents

**class** evalml.data_checks.no_variance_data_check.**NoVarianceDataCheck**(*count_nan_as_value=False*)

Check if the target or any of the features have no variance.

**Parameters** **count_nan_as_value** (*bool*) – If True, missing values will be counted as their own unique value. Additionally, if true, will return a DataCheckWarning instead of an error if the feature has mostly missing data and only one unique value. Defaults to False.

**Methods**

| | |
|---|---|
| *name* | Return a name describing the data check. |
| *validate* | Check if the target or any of the features have no variance (1 unique value). |

**name**(*cls*)

Return a name describing the data check.

**validate**(*self, X, y=None*)

Check if the target or any of the features have no variance (1 unique value).

**Parameters**

- **X** (*pd.DataFrame, np.ndarray*) – The input features.
- **y** (*pd.Series, np.ndarray*) – Optional, the target data.

**Returns** A dict of warnings/errors corresponding to features or target with no variance.

**Return type** dict

#### Examples

```
>>> import pandas as pd
```

Columns or target data that have only one unique value will raise an error.

```
>>> X = pd.DataFrame([2, 2, 2, 2, 2, 2, 2, 2], columns=["First_Column"])
>>> y = pd.Series([1, 1, 1, 1, 1, 1, 1, 1])
...
```

(continues on next page)

(continued from previous page)

```
>>> novar_dc = NoVarianceDataCheck()
>>> assert novar_dc.validate(X, y) == [
...     {
...         "message": "'First_Column' has 1 unique value.",
...         "data_check_name": "NoVarianceDataCheck",
...         "level": "warning",
...         "details": {"columns": ["First_Column"], "rows": None},
...         "code": "NO_VARIANCE",
...         "action_options": [
...             {
...                 "code": "DROP_COL",
...                 "data_check_name": "NoVarianceDataCheck",
...                 "parameters": {},
...                 "metadata": {"columns": ["First_Column"], "rows": None}
...             },
...         ]
...     },
...     {
...         "message": "Y has 1 unique value.",
...         "data_check_name": "NoVarianceDataCheck",
...         "level": "warning",
...         "details": {"columns": ["Y"], "rows": None},
...         "code": "NO_VARIANCE",
...         "action_options": []
...     }
... ]
```

By default, NaNs will not be counted as distinct values. In the first example, there are still two distinct values besides None. In the second, there are no distinct values as the target is entirely null.

```
>>> X["First_Column"] = [2, 2, 2, 3, 3, 3, None, None]
>>> y = pd.Series([1, 1, 1, 2, 2, 2, None, None])
>>> assert novar_dc.validate(X, y) == []
...
...
>>> y = pd.Series([None] * 7)
>>> assert novar_dc.validate(X, y) == [
...     {
...         "message": "Y has 0 unique values.",
...         "data_check_name": "NoVarianceDataCheck",
...         "level": "warning",
...         "details": {"columns": ["Y"], "rows": None},
...         "code": "NO_VARIANCE_ZERO_UNIQUE",
...         "action_options":[]
...     }
... ]
```

As None is not considered a distinct value by default, there is only one unique value in X and y.

```
>>> X["First_Column"] = [2, 2, 2, 2, None, None, None, None]
>>> y = pd.Series([1, 1, 1, 1, None, None, None, None])
>>> assert novar_dc.validate(X, y) == [
```

(continues on next page)

(continued from previous page)

```
...     {
...         "message": "'First_Column' has 1 unique value.",
...         "data_check_name": "NoVarianceDataCheck",
...         "level": "warning",
...         "details": {"columns": ["First_Column"], "rows": None},
...         "code": "NO_VARIANCE",
...         "action_options": [
...             {
...                 "code": "DROP_COL",
...                  "data_check_name": "NoVarianceDataCheck",
...                  "parameters": {},
...                  "metadata": {"columns": ["First_Column"], "rows": None}
...             },
...         ]
...     },
...     {
...         "message": "Y has 1 unique value.",
...         "data_check_name": "NoVarianceDataCheck",
...         "level": "warning",
...         "details": {"columns": ["Y"], "rows": None},
...         "code": "NO_VARIANCE",
...         "action_options": []
...     }
... ]
```

If count_nan_as_value is set to True, then NaNs are counted as unique values. In the event that there is an adequate number of unique values only because count_nan_as_value is set to True, a warning will be raised so the user can encode these values.

```
>>> novar_dc = NoVarianceDataCheck(count_nan_as_value=True)
>>> assert novar_dc.validate(X, y) == [
...     {
...         "message": "'First_Column' has two unique values including nulls.␣
↪Consider encoding the nulls for this column to be useful for machine learning.
↪",
...         "data_check_name": "NoVarianceDataCheck",
...         "level": "warning",
...         "details": {"columns": ["First_Column"], "rows": None},
...         "code": "NO_VARIANCE_WITH_NULL",
...         "action_options": [
...             {
...                 "code": "DROP_COL",
...                  "data_check_name": "NoVarianceDataCheck",
...                  "parameters": {},
...                  "metadata": {"columns": ["First_Column"], "rows": None}
...             },
...         ]
...     },
...     {
...         "message": "Y has two unique values including nulls. Consider␣
↪encoding the nulls for this column to be useful for machine learning.",
...         "data_check_name": "NoVarianceDataCheck",
```

(continues on next page)

(continued from previous page)

```
...         "level": "warning",
...         "details": {"columns": ["Y"], "rows": None},
...         "code": "NO_VARIANCE_WITH_NULL",
...         "action_options": []
...     }
... ]
```

### null_data_check

Data check that checks if there are any highly-null columns and rows in the input.

### Module Contents

### Classes Summary

| | |
|---|---|
| *NullDataCheck* | Check if there are any highly-null numerical, boolean, categorical, natural language, and unknown columns and rows in the input. |

### Contents

**class** evalml.data_checks.null_data_check.**NullDataCheck**(*pct_null_col_threshold=0.95*, *pct_moderately_null_col_threshold=0.2*, *pct_null_row_threshold=0.95*)

Check if there are any highly-null numerical, boolean, categorical, natural language, and unknown columns and rows in the input.

**Parameters**

- **pct_null_col_threshold** (*float*) – If the percentage of NaN values in an input feature exceeds this amount, that column will be considered highly-null. Defaults to 0.95.
- **pct_moderately_null_col_threshold** (*float*) – If the percentage of NaN values in an input feature exceeds this amount but is less than the percentage specified in pct_null_col_threshold, that column will be considered moderately-null. Defaults to 0.20.
- **pct_null_row_threshold** (*float*) – If the percentage of NaN values in an input row exceeds this amount, that row will be considered highly-null. Defaults to 0.95.

**Methods**

| | |
|---|---|
| *get_null_column_information* | Finds columns that are considered highly null (percentage null is greater than threshold) and returns dictionary mapping column name to percentage null and dictionary mapping column name to null indices. |
| *get_null_row_information* | Finds rows that are considered highly null (percentage null is greater than threshold). |
| *name* | Return a name describing the data check. |
| *validate* | Check if there are any highly-null columns or rows in the input. |

**static get_null_column_information**(*X*, *pct_null_col_threshold=0.0*)

Finds columns that are considered highly null (percentage null is greater than threshold) and returns dictionary mapping column name to percentage null and dictionary mapping column name to null indices.

**Parameters**

- **X** (*pd.DataFrame*) – DataFrame to check for highly null columns.
- **pct_null_col_threshold** (*float*) – Percentage threshold for a column to be considered null. Defaults to 0.0.

**Returns** Tuple containing: dictionary mapping column name to its null percentage and dictionary mapping column name to null indices in that column.

**Return type** tuple

**static get_null_row_information**(*X*, *pct_null_row_threshold=0.0*)

Finds rows that are considered highly null (percentage null is greater than threshold).

**Parameters**

- **X** (*pd.DataFrame*) – DataFrame to check for highly null rows.
- **pct_null_row_threshold** (*float*) – Percentage threshold for a row to be considered null. Defaults to 0.0.

**Returns** Series containing the percentage null for each row.

**Return type** pd.Series

**name**(*cls*)

Return a name describing the data check.

**validate**(*self*, *X*, *y=None*)

Check if there are any highly-null columns or rows in the input.

**Parameters**

- **X** (*pd.DataFrame, np.ndarray*) – Features.
- **y** (*pd.Series, np.ndarray*) – Ignored. Defaults to None.

**Returns** dict with a DataCheckWarning if there are any highly-null columns or rows.

**Return type** dict

### Examples

```
>>> import pandas as pd
...
>>> class SeriesWrap():
...     def __init__(self, series):
...         self.series = series
...
...     def __eq__(self, series_2):
...         return all(self.series.eq(series_2.series))
```

With pct_null_col_threshold set to 0.50, any column that has 50% or more of its observations set to null will be included in the warning, as well as the percentage of null values identified ("all_null": 1.0, "lots_of_null": 0.8).

```
>>> df = pd.DataFrame({
...     "all_null": [None, pd.NA, None, None, None],
...     "lots_of_null": [None, None, None, None, 5],
...     "few_null": [1, 2, None, 2, 3],
...     "no_null": [1, 2, 3, 4, 5]
... })
...
>>> highly_null_dc = NullDataCheck(pct_null_col_threshold=0.50)
>>> assert highly_null_dc.validate(df) == [
...     {
...         "message": "Column(s) 'all_null', 'lots_of_null' are 50.0% or more␣
↪null",
...         "data_check_name": "NullDataCheck",
...         "level": "warning",
...         "details": {
...             "columns": ["all_null", "lots_of_null"],
...             "rows": None,
...             "pct_null_rows": {"all_null": 1.0, "lots_of_null": 0.8}
...         },
...         "code": "HIGHLY_NULL_COLS",
...         "action_options": [
...             {
...                 "code": "DROP_COL",
...                 "data_check_name": "NullDataCheck",
...                 "parameters": {},
...                 "metadata": {"columns": ["all_null", "lots_of_null"], "rows
↪": None}
...             }
...         ]
...     },
...     {
...         "message": "Column(s) 'few_null' have between 20.0% and 50.0% null␣
↪values",
...         "data_check_name": "NullDataCheck",
...         "level": "warning",
...         "details": {"columns": ["few_null"], "rows": None},
...         "code": "COLS_WITH_NULL",
...         "action_options": [
...             {
...                 "code": "IMPUTE_COL",
...                 "data_check_name": "NullDataCheck",
...                 "metadata": {"columns": ["few_null"], "rows": None, "is_
↪target": False},
...                 "parameters": {
...                     "impute_strategies": {
...                         "parameter_type": "column",
...                         "columns": {
...                             "few_null": {
...                                 "impute_strategy": {"categories": ["mean",
↪"most_frequent"], "type": "category", "default_value": "mean"}
...                             }
...                         }
...                     }
```

(continues on next page)

(continued from previous page)

```
...                 }
...             }
...         ]
...     }
... ]
```

With pct_null_row_threshold set to 0.50, any row with 50% or more of its respective column values set to null will included in the warning, as well as the offending rows ("rows": [0, 1, 2, 3]). Since the default value for pct_null_col_threshold is 0.95, "all_null" is also included in the warnings since the percentage of null values in that row is over 95%. Since the default value for pct_moderately_null_col_threshold is 0.20, "few_null" is included as a "moderately null" column as it has a null column percentage of 20%.

```
>>> highly_null_dc = NullDataCheck(pct_null_row_threshold=0.50)
>>> validation_messages = highly_null_dc.validate(df)
>>> validation_messages[0]["details"]["pct_null_cols"] = SeriesWrap(validation_
↪messages[0]["details"]["pct_null_cols"])
>>> highly_null_rows = SeriesWrap(pd.Series([0.5, 0.5, 0.75, 0.5]))
>>> assert validation_messages == [
...     {
...         "message": "4 out of 5 rows are 50.0% or more null",
...         "data_check_name": "NullDataCheck",
...         "level": "warning",
...         "details": {
...             "columns": None,
...             "rows": [0, 1, 2, 3],
...             "pct_null_cols": highly_null_rows
...         },
...         "code": "HIGHLY_NULL_ROWS",
...         "action_options": [
...             {
...                 "code": "DROP_ROWS",
...                  "data_check_name": "NullDataCheck",
...                  "parameters": {},
...                  "metadata": {"columns": None, "rows": [0, 1, 2, 3]}
...              }
...         ]
...     },
...     {
...         "message": "Column(s) 'all_null' are 95.0% or more null",
...         "data_check_name": "NullDataCheck",
...         "level": "warning",
...         "details": {
...             "columns": ["all_null"],
...             "rows": None,
...             "pct_null_rows": {"all_null": 1.0}
...         },
...        "code": "HIGHLY_NULL_COLS",
...        "action_options": [
...             {
...                 "code": "DROP_COL",
...                 "data_check_name": "NullDataCheck",
...                 "metadata": {"columns": ["all_null"], "rows": None},
```

(continues on next page)

(continued from previous page)

```
...                 "parameters": {}
...             }
...         ]
...     },
...     {
...         "message": "Column(s) 'lots_of_null', 'few_null' have between 20.0%␣
↪and 95.0% null values",
...         "data_check_name": "NullDataCheck",
...         "level": "warning",
...         "details": {"columns": ["lots_of_null", "few_null"], "rows": None},
...         "code": "COLS_WITH_NULL",
...         "action_options": [
...             {
...                "code": "IMPUTE_COL",
...                "data_check_name": "NullDataCheck",
...                "metadata": {"columns": ["lots_of_null", "few_null"], "rows":
↪ None, "is_target": False},
...                "parameters": {
...                    "impute_strategies": {
...                        "parameter_type": "column",
...                        "columns": {
...                            "lots_of_null": {"impute_strategy": {"categories
↪": ["mean", "most_frequent"], "type": "category", "default_value": "mean"}},
...                            "few_null": {"impute_strategy": {"categories": [
↪"mean", "most_frequent"], "type": "category", "default_value": "mean"}}
...                        }
...                    }
...                }
...             }
...         ]
...     }
... ]
```

### outliers_data_check

Data check that checks if there are any outliers in input data by using IQR to determine score anomalies.

### Module Contents

### Classes Summary

| | |
|---|---|
| *OutliersDataCheck* | Checks if there are any outliers in input data by using IQR to determine score anomalies. |

## Contents

**class** evalml.data_checks.outliers_data_check.**OutliersDataCheck**

Checks if there are any outliers in input data by using IQR to determine score anomalies.

Columns with score anomalies are considered to contain outliers.

**Methods**

| | |
|---|---|
| *get_boxplot_data* | Returns box plot information for the given data. |
| *name* | Return a name describing the data check. |
| *validate* | Check if there are any outliers in a dataframe by using IQR to determine column anomalies. Column with anomalies are considered to contain outliers. |

**static get_boxplot_data**(*data_*)

Returns box plot information for the given data.

**Parameters** **data** (*pd.Series, np.ndarray*) – Input data.

**Returns** A payload of box plot statistics.

**Return type** dict

### Examples

```
>>> import pandas as pd
...
>>> df = pd.DataFrame({
...     "x": [1, 2, 3, 4, 5],
...     "y": [6, 7, 8, 9, 10],
...     "z": [-1, -2, -3, -1201, -4]
... })
>>> box_plot_data = OutliersDataCheck.get_boxplot_data(df["z"])
>>> box_plot_data["score"] = round(box_plot_data["score"], 2)
>>> assert box_plot_data == {
...     "score": 0.89,
...     "pct_outliers": 0.2,
...     "values": {"q1": -4.0,
...                "median": -3.0,
...                "q3": -2.0,
...                "low_bound": -7.0,
...                "high_bound": -1.0,
...                "low_values": [-1201],
...                "high_values": [],
...                "low_indices": [3],
...                "high_indices": []}
...     }
```

**name**(*cls*)

Return a name describing the data check.

**validate**(*self*, *X*, *y=None*)

Check if there are any outliers in a dataframe by using IQR to determine column anomalies. Column with anomalies are considered to contain outliers.

**Parameters**

- **X** (*pd.DataFrame, np.ndarray*) – Input features.
- **y** (*pd.Series, np.ndarray*) – Ignored. Defaults to None.

**Returns** A dictionary with warnings if any columns have outliers.

**Return type** dict

### Examples

```
>>> import pandas as pd
```

The column "z" has an outlier so a warning is added to alert the user of its location.

```
>>> df = pd.DataFrame({
...     "x": [1, 2, 3, 4, 5],
...     "y": [6, 7, 8, 9, 10],
...     "z": [-1, -2, -3, -1201, -4]
... })
...
>>> outliers_check = OutliersDataCheck()
>>> assert outliers_check.validate(df) == [
...     {
...         "message": "Column(s) 'z' are likely to have outlier data.",
...         "data_check_name": "OutliersDataCheck",
...         "level": "warning",
...         "code": "HAS_OUTLIERS",
...         "details": {"columns": ["z"], "rows": [3], "column_indices": {"z":␣
↪[3]}},
...         "action_options": [
...             {
...                 "code": "DROP_ROWS",
...                  "data_check_name": "OutliersDataCheck",
...                  "parameters": {},
...                  "metadata": {"rows": [3], "columns": None}
...             }
...         ]
...     }
... ]
```

### sparsity_data_check

Data check that checks if there are any columns with sparsely populated values in the input.

### Module Contents

### Classes Summary

| | |
|---|---|
| *SparsityDataCheck* | Check if there are any columns with sparsely populated values in the input. |

### Attributes Summary

| |
|---|
| *warning_too_unique* |

### Contents

**class** evalml.data_checks.sparsity_data_check.**SparsityDataCheck**(*problem_type*, *threshold*, *unique_count_threshold=10*)

Check if there are any columns with sparsely populated values in the input.

**Parameters**

- **problem_type** (*str or ProblemTypes*) – The specific problem type to data check for. 'multiclass' or 'time series multiclass' is the only accepted problem type.
- **threshold** (*float*) – The threshold value, or percentage of each column's unique values, below which, a column exhibits sparsity. Should be between 0 and 1.
- **unique_count_threshold** (*int*) – The minimum number of times a unique value has to be present in a column to not be considered "sparse." Defaults to 10.

**Methods**

| | |
|---|---|
| *name* | Return a name describing the data check. |
| *sparsity_score* | Calculate a sparsity score for the given value counts by calculating the percentage of unique values that exceed the count_threshold. |
| *validate* | Calculate what percentage of each column's unique values exceed the count threshold and compare that percentage to the sparsity threshold stored in the class instance. |

**name**(*cls*)

Return a name describing the data check.

**static sparsity_score**(*col*, *count_threshold=10*)

Calculate a sparsity score for the given value counts by calculating the percentage of unique values that exceed the count_threshold.

**Parameters**

- **col** (*pd.Series*) – Feature values.
- **count_threshold** (*int*) – The number of instances below which a value is considered sparse. Default is 10.

**Returns** Sparsity score, or the percentage of the unique values that exceed count_threshold.

**Return type** (float)

**validate**(*self, X, y=None*)

Calculate what percentage of each column's unique values exceed the count threshold and compare that percentage to the sparsity threshold stored in the class instance.

**Parameters**

- **X** (*pd.DataFrame, np.ndarray*) – Features.
- **y** (*pd.Series, np.ndarray*) – Ignored.

**Returns** dict with a DataCheckWarning if there are any sparse columns.

**Return type** dict

### Examples

```
>>> import pandas as pd
```

For multiclass problems, if a column doesn't have enough representation from unique values, it will be considered sparse.

```
>>> df = pd.DataFrame({
...     "sparse": [float(x) for x in range(100)],
...     "not_sparse": [float(1) for x in range(100)]
... })
...
>>> sparsity_check = SparsityDataCheck(problem_type="multiclass", threshold=0.5,
↪ unique_count_threshold=10)
>>> assert sparsity_check.validate(df) == [
...     {
...         "message": "Input columns ('sparse') for multiclass problem type␣
↪are too sparse.",
...         "data_check_name": "SparsityDataCheck",
...         "level": "warning",
...         "code": "TOO_SPARSE",
...         "details": {
...             "columns": ["sparse"],
...             "sparsity_score": {"sparse": 0.0},
...             "rows": None
...         },
...         "action_options": [
...             {
...                 "code": "DROP_COL",
...                  "data_check_name": "SparsityDataCheck",
...                  "parameters": {},
...                  "metadata": {"columns": ["sparse"], "rows": None}
```

(continues on next page)

(continued from previous page)

```
...                 }
...             ]
...         }
... ]
```

... >>> df["sparse"] = [float(x % 10) for x in range(100)] >>> sparsity_check = SparsityDataCheck(problem_type="multiclass", threshold=1, unique_count_threshold=5) >>> assert sparsity_check.validate(df) == [] ... >>> sparse_array = pd.Series([1, 1, 1, 2, 2, 3] * 3) >>> assert SparsityDataCheck.sparsity_score(sparse_array, count_threshold=5) == 0.6666666666666666

`evalml.data_checks.sparsity_data_check.`**`warning_too_unique = Input columns ({}) for {} problem type are too sparse.`**

### target_distribution_data_check

Data check that checks if the target data contains certain distributions that may need to be transformed prior training to improve model performance.

### Module Contents

### Classes Summary

| | |
|---|---|
| *TargetDistributionDataCheck* | Check if the target data contains certain distributions that may need to be transformed prior training to improve model performance. Uses the Shapiro-Wilks test when the dataset is <=5000 samples, otherwise uses Jarque-Bera. |

### Contents

**class** `evalml.data_checks.target_distribution_data_check.`**`TargetDistributionDataCheck`**

Check if the target data contains certain distributions that may need to be transformed prior training to improve model performance. Uses the Shapiro-Wilks test when the dataset is <=5000 samples, otherwise uses Jarque-Bera.

**Methods**

| | |
|---|---|
| *name* | Return a name describing the data check. |
| *validate* | Check if the target data has a certain distribution. |

**name**(*cls*)

Return a name describing the data check.

**validate**(*self*, *X*, *y*)

Check if the target data has a certain distribution.

**Parameters**

- **X** (*pd.DataFrame, np.ndarray*) – Features. Ignored.

- **y** (*pd.Series, np.ndarray*) – Target data to check for underlying distributions.

**Returns** List with DataCheckErrors if certain distributions are found in the target data.

**Return type** dict (DataCheckError)

### Examples

```
>>> import pandas as pd
```

Targets that exhibit a lognormal distribution will raise a warning for the user to transform the target.

```
>>> y = [0.946, 0.972, 1.154, 0.954, 0.969, 1.222, 1.038, 0.999, 0.973, 0.897]
>>> target_check = TargetDistributionDataCheck()
>>> assert target_check.validate(None, y) == [
...     {
...         "message": "Target may have a lognormal distribution.",
...         "data_check_name": "TargetDistributionDataCheck",
...         "level": "warning",
...         "code": "TARGET_LOGNORMAL_DISTRIBUTION",
...         "details": {"normalization_method": "shapiro", "statistic": 0.8, "p-
↪value": 0.045, "columns": None, "rows": None},
...         "action_options": [
...             {
...                 "code": "TRANSFORM_TARGET",
...                 "data_check_name": "TargetDistributionDataCheck",
...                 "parameters": {},
...                 "metadata": {
...                     "transformation_strategy": "lognormal",
...                     "is_target": True,
...                     "columns": None,
...                     "rows": None
...                 }
...             }
...         ]
...     }
... ]
...
>>> y = pd.Series([1, 1, 1, 2, 2, 3, 4, 4, 5, 5, 5])
>>> assert target_check.validate(None, y) == []
...
...
>>> y = pd.Series(pd.date_range("1/1/21", periods=10))
>>> assert target_check.validate(None, y) == [
...     {
...         "message": "Target is unsupported datetime type. Valid Woodwork␣
↪logical types include: integer, double, age, age_fractional",
...         "data_check_name": "TargetDistributionDataCheck",
...         "level": "error",
...         "details": {"columns": None, "rows": None, "unsupported_type":
↪"datetime"},
...         "code": "TARGET_UNSUPPORTED_TYPE",
...         "action_options": []
```

(continues on next page)

(continued from previous page)

```
...     }
... ]
```

### target_leakage_data_check

Data check that checks if any of the features are highly correlated with the target by using mutual information or Pearson correlation.

### Module Contents

### Classes Summary

| | |
|---|---|
| *TargetLeakageDataCheck* | Check if any of the features are highly correlated with the target by using mutual information, Pearson correlation, and other correlation metrics. |

### Contents

**class** evalml.data_checks.target_leakage_data_check.**TargetLeakageDataCheck**(*pct_corr_threshold=0.95*, *method='all'*)

Check if any of the features are highly correlated with the target by using mutual information, Pearson correlation, and other correlation metrics.

If method='mutual_info', this data check uses mutual information and supports all target and feature types. Other correlation metrics only support binary with numeric and boolean dtypes. This method will return a value in [-1, 1] if other correlation metrics are selected and will returns a value in [0, 1] if mutual information is selected. Correlation metrics available can be found in Woodwork's dependence_dict method.

**Parameters**

- **pct_corr_threshold** (*float*) – The correlation threshold to be considered leakage. Defaults to 0.95.
- **method** (*string*) – The method to determine correlation. Use 'all' or 'max' for the maximum correlation, or for specific correlation metrics, use their name (ie 'mutual_info' for mutual information, 'pearson' for Pearson correlation, etc). possible methods can be found in Woodwork's config, under *correlation_metrics*. Defaults to 'all'.

**Methods**

| | |
|---|---|
| *name* | Return a name describing the data check. |
| *validate* | Check if any of the features are highly correlated with the target by using mutual information, Pearson correlation, and/or Spearman correlation. |

**name**(*cls*)

Return a name describing the data check.

**validate**(*self*, *X*, *y*)

Check if any of the features are highly correlated with the target by using mutual information, Pearson correlation, and/or Spearman correlation.

If *method='mutual_info'* or *'method='max'*, supports all target and feature types. Other correlation metrics only support binary with numeric and boolean dtypes. This method will return a value in [-1, 1] if other correlation metrics are selected and will returns a value in [0, 1] if mutual information is selected.

**Parameters**

- **X** (*pd.DataFrame, np.ndarray*) – The input features to check.
- **y** (*pd.Series, np.ndarray*) – The target data.

**Returns** dict with a DataCheckWarning if target leakage is detected.

**Return type** dict (DataCheckWarning)

### Examples

```
>>> import pandas as pd
```

Any columns that are strongly correlated with the target will raise a warning. This could be indicative of data leakage.

```
>>> X = pd.DataFrame({
...     "leak": [10, 42, 31, 51, 61] * 15,
...     "x": [42, 54, 12, 64, 12] * 15,
...     "y": [13, 5, 13, 74, 24] * 15,
... })
>>> y = pd.Series([10, 42, 31, 51, 40] * 15)
...
>>> target_leakage_check = TargetLeakageDataCheck(pct_corr_threshold=0.95)
>>> assert target_leakage_check.validate(X, y) == [
...     {
...         "message": "Column 'leak' is 95.0% or more correlated with the␣
↪target",
...         "data_check_name": "TargetLeakageDataCheck",
...         "level": "warning",
...         "code": "TARGET_LEAKAGE",
...         "details": {"columns": ["leak"], "rows": None},
...         "action_options": [
...             {
...                 "code": "DROP_COL",
...                 "data_check_name": "TargetLeakageDataCheck",
...                 "parameters": {},
...                 "metadata": {"columns": ["leak"], "rows": None}
...             }
...         ]
...     }
... ]
```

The default method can be changed to pearson from mutual_info.

```
>>> X["x"] = y / 2
>>> target_leakage_check = TargetLeakageDataCheck(pct_corr_threshold=0.8,␣
↪method="pearson")
>>> assert target_leakage_check.validate(X, y) == [
...     {
...         "message": "Columns 'leak', 'x' are 80.0% or more correlated with␣
↪the target",
...         "data_check_name": "TargetLeakageDataCheck",
...         "level": "warning",
...         "details": {"columns": ["leak", "x"], "rows": None},
...         "code": "TARGET_LEAKAGE",
...         "action_options": [
...             {
...                 "code": "DROP_COL",
...                  "data_check_name": "TargetLeakageDataCheck",
...                  "parameters": {},
...                  "metadata": {"columns": ["leak", "x"], "rows": None}
...             }
...         ]
...     }
... ]
```

### ts_parameters_data_check

Data check that checks whether the time series parameters are compatible with the data size.

### Module Contents

### Classes Summary

| | |
|---|---|
| *TimeSeriesParametersDataCheck* | Checks whether the time series parameters are compatible with data splitting. |

### Contents

**class** evalml.data_checks.ts_parameters_data_check.**TimeSeriesParametersDataCheck**(*problem_configuration*, *n_splits*)

Checks whether the time series parameters are compatible with data splitting.

If *gap + max_delay + forecast_horizon > X.shape[0] // (n_splits + 1)*

then the feature engineering window is larger than the smallest split. This will cause the pipeline to create features from data that does not exist, which will cause errors.

**Parameters**

- **problem_configuration** (*dict*) – Dict containing problem_configuration parameters.
- **n_splits** (*int*) – Number of time series splits.

**Methods**

| | |
|---|---|
| *name* | Return a name describing the data check. |
| *validate* | Check if the time series parameters are compatible with data splitting. |

**name**(*cls*)

Return a name describing the data check.

**validate**(*self*, *X*, *y=None*)

Check if the time series parameters are compatible with data splitting.

**Parameters**

- **X** (*pd.DataFrame, np.ndarray*) – Features.
- **y** (*pd.Series, np.ndarray*) – Ignored. Defaults to None.

**Returns** dict with a DataCheckError if parameters are too big for the split sizes.

**Return type** dict

### Examples

```
>>> import pandas as pd
```

The time series parameters have to be compatible with the data passed. If the window size (gap + max_delay + forecast_horizon) is greater than or equal to the split size, then an error will be raised.

```
>>> X = pd.DataFrame({
...     "dates": pd.date_range("1/1/21", periods=100),
...     "first": [i for i in range(100)],
... })
>>> y = pd.Series([i for i in range(100)])
...
>>> problem_config = {"gap": 7, "max_delay": 2, "forecast_horizon": 12, "time_
↪index": "dates"}
>>> ts_parameters_check = TimeSeriesParametersDataCheck(problem_
↪configuration=problem_config, n_splits=7)
>>> assert ts_parameters_check.validate(X, y) == [
...     {
...         "message": "Since the data has 100 observations, n_splits=7, and a␣
↪forecast horizon of 12, the smallest "
...                    "split would have 16 observations. Since 21 (gap + max_
↪delay + forecast_horizon)"
...                    " >= 16, then at least one of the splits would be empty␣
↪by the time it reaches "
...                    "the pipeline. Please use a smaller number of splits,␣
↪reduce one or more these "
...                    "parameters, or collect more data.",
...         "data_check_name": "TimeSeriesParametersDataCheck",
...         "level": "error",
...         "code": "TIMESERIES_PARAMETERS_NOT_COMPATIBLE_WITH_SPLIT",
...         "details": {
```

(continues on next page)

(continued from previous page)

```
...             "columns": None,
...             "rows": None,
...             "max_window_size": 21,
...             "min_split_size": 16,
...             "n_obs": 100,
...             "n_splits": 7
...         },
...         "action_options": []
...     }
... ]
```

### ts_splitting_data_check

Data check that checks whether the time series training and validation splits have adequate class representation.

### Module Contents

### Classes Summary

| | |
|---|---|
| *TimeSeriesSplittingDataCheck* | Checks whether the time series target data is compatible with splitting. |

### Contents

**class** evalml.data_checks.ts_splitting_data_check.**TimeSeriesSplittingDataCheck**(*problem_type*, *n_splits*)

Checks whether the time series target data is compatible with splitting.

If the target data in the training and validation of every split doesn't have representation from all classes (for time series classification problems) this will prevent the estimators from training on all potential outcomes which will cause errors during prediction.

**Parameters**

- **problem_type** (*str or ProblemTypes*) – Problem type.
- **n_splits** (*int*) – Number of time series splits.

**Methods**

| | |
|---|---|
| *name* | Return a name describing the data check. |
| *validate* | Check if the training and validation targets are compatible with time series data splitting. |

**name**(*cls*)

Return a name describing the data check.

**validate**(*self*, *X*, *y*)

Check if the training and validation targets are compatible with time series data splitting.

**Parameters**

- **X** (*pd.DataFrame, np.ndarray*) – Ignored. Features.
- **y** (*pd.Series, np.ndarray*) – Target data.

**Returns** dict with a DataCheckError if splitting would result in inadequate class representation.

**Return type** dict

### Example

```
>>> import pandas as pd
```

Passing n_splits as 3 means that the data will be segmented into 4 parts to be iterated over for training and validation splits. The first split results in training indices of [0:25] and validation indices of [25:50]. The training indices of the first split result in only one unique value (0). The third split results in training indices of [0:75] and validation indices of [75:100]. The validation indices of the third split result in only one unique value (1).

```
>>> X = None
>>> y = pd.Series([0 if i < 45 else i % 2 if i < 55 else 1 for i in range(100)])
>>> ts_splitting_check = TimeSeriesSplittingDataCheck("time series binary", 3)
>>> assert ts_splitting_check.validate(X, y) == [
...     {
...         "message": "Time Series Binary and Time Series Multiclass problem "
...                    "types require every training and validation split to "
...                    "have at least one instance of all the target classes. "
...                    "The following splits are invalid: [1, 3]",
...         "data_check_name": "TimeSeriesSplittingDataCheck",
...         "level": "error",
...         "details": {
...             "columns": None, "rows": None,
...             "invalid_splits": {
...                 1: {"Training": [0, 25]},
...                 3: {"Validation": [75, 100]}
...             }
...         },
...         "code": "TIMESERIES_TARGET_NOT_COMPATIBLE_WITH_SPLIT",
...         "action_options": []
...     }
... ]
```

### uniqueness_data_check

Data check that checks if there are any columns in the input that are either too unique for classification problems or not unique enough for regression problems.

#### Module Contents

#### Classes Summary

| | |
|---|---|
| *UniquenessDataCheck* | Check if there are any columns in the input that are either too unique for classification problems or not unique enough for regression problems. |

#### Attributes Summary

| |
|---|
| *warning_not_unique_enough* |
| *warning_too_unique* |

#### Contents

**class** evalml.data_checks.uniqueness_data_check.**UniquenessDataCheck**(*problem_type*, *threshold=0.5*)

Check if there are any columns in the input that are either too unique for classification problems or not unique enough for regression problems.

**Parameters**

- **problem_type** (*str or ProblemTypes*) – The specific problem type to data check for. e.g. 'binary', 'multiclass', 'regression, 'time series regression'
- **threshold** (*float*) – The threshold to set as an upper bound on uniqueness for classification type problems or lower bound on for regression type problems. Defaults to 0.50.

**Methods**

| | |
|---|---|
| *name* | Return a name describing the data check. |
| *uniqueness_score* | Calculate a uniqueness score for the provided field. NaN values are not considered as unique values in the calculation. |
| *validate* | Check if there are any columns in the input that are too unique in the case of classification problems or not unique enough in the case of regression problems. |

**name**(*cls*)

Return a name describing the data check.

**static uniqueness_score**(*col*, *drop_na=True*)

Calculate a uniqueness score for the provided field. NaN values are not considered as unique values in the calculation.

Based on the Herfindahl-Hirschman Index.

**Parameters**

- **col** (*pd.Series*) – Feature values.
- **drop_na** (*bool*) – Whether to drop null values when computing the uniqueness score. Defaults to True.

**Returns** Uniqueness score.

**Return type** (float)

**validate**(*self*, *X*, *y=None*)

Check if there are any columns in the input that are too unique in the case of classification problems or not unique enough in the case of regression problems.

**Parameters**

- **X** (*pd.DataFrame, np.ndarray*) – Features.
- **y** (*pd.Series, np.ndarray*) – Ignored. Defaults to None.

**Returns**

**dict with a DataCheckWarning if there are any too unique or not** unique enough columns.

**Return type** dict

### Examples

```
>>> import pandas as pd
```

Because the problem type is regression, the column "regression_not_unique_enough" raises a warning for having just one value.

```
>>> df = pd.DataFrame({
...     "regression_unique_enough": [float(x) for x in range(100)],
...     "regression_not_unique_enough": [float(1) for x in range(100)]
... })
...
>>> uniqueness_check = UniquenessDataCheck(problem_type="regression",␣
↪threshold=0.8)
>>> assert uniqueness_check.validate(df) == [
...     {
...         "message": "Input columns 'regression_not_unique_enough' for␣
↪regression problem type are not unique enough.",
...         "data_check_name": "UniquenessDataCheck",
...         "level": "warning",
...         "code": "NOT_UNIQUE_ENOUGH",
...         "details": {"columns": ["regression_not_unique_enough"],
↪"uniqueness_score": {"regression_not_unique_enough": 0.0}, "rows": None},
...         "action_options": [
```

(continues on next page)

(continued from previous page)

```
...             {
...                 "code": "DROP_COL",
...                 "parameters": {},
...                 "data_check_name": "UniquenessDataCheck",
...                 "metadata": {"columns": ["regression_not_unique_enough"],
↪"rows": None}
...             }
...         ]
...     }
... ]
```

For multiclass, the column "regression_unique_enough" has too many unique values and will raise an appropriate warning. >>> y = pd.Series([1, 1, 1, 2, 2, 3, 3, 3]) >>> uniqueness_check = UniquenessDataCheck(problem_type="multiclass", threshold=0.8) >>> assert uniqueness_check.validate(df) == [ … { … "message": "Input columns 'regression_unique_enough' for multiclass problem type are too unique.", … "data_check_name": "UniquenessDataCheck", … "level": "warning", … "details": { … "columns": ["regression_unique_enough"], … "rows": None, … "uniqueness_score": {"regression_unique_enough": 0.99} … }, … "code": "TOO_UNIQUE", … "action_options": [ … { … "code": "DROP_COL", … "data_check_name": "UniquenessDataCheck", … "parameters": {}, … "metadata": {"columns": ["regression_unique_enough"], "rows": None} … } … ] … } … ] … >>> assert UniquenessDataCheck.uniqueness_score(y) == 0.65625

evalml.data_checks.uniqueness_data_check.**warning_not_unique_enough = Input columns {} for {} problem type are not unique enough.**

evalml.data_checks.uniqueness_data_check.**warning_too_unique = Input columns {} for {} problem type are too unique.**

### utils

Utility methods for the data checks in EvalML.

### Module Contents

### Functions

| | |
|---|---|
| *handle_data_check_action_code* | Handles data check action codes by either returning the DataCheckActionCode or converting from a str. |

### Contents

evalml.data_checks.utils.**handle_data_check_action_code**(*action_code*)

Handles data check action codes by either returning the DataCheckActionCode or converting from a str.

**Parameters** **action_code** (*str or DataCheckActionCode*) – Data check action code that needs to be handled.

**Returns** DataCheckActionCode enum

**Raises**

- **KeyError** – If input is not a valid DataCheckActionCode enum value.
- **ValueError** – If input is not a string or DatCheckActionCode object.

### Examples

```
>>> assert handle_data_check_action_code("drop_col") == DataCheckActionCode.DROP_COL
>>> assert handle_data_check_action_code("DROP_ROWS") == DataCheckActionCode.DROP_
↪ROWS
>>> assert handle_data_check_action_code("Impute_col") == DataCheckActionCode.
↪IMPUTE_COL
```

## Package Contents

### Classes Summary

| | |
|---|---|
| *ClassImbalanceDataCheck* | Check if any of the target labels are imbalanced, or if the number of values for each target are below 2 times the number of CV folds. Use for classification problems. |
| *DataCheck* | Base class for all data checks. |
| *DataCheckAction* | A recommended action returned by a DataCheck. |
| *DataCheckActionCode* | Enum for data check action code. |
| *DataCheckActionOption* | A recommended action option returned by a DataCheck. |
| *DataCheckError* | DataCheckMessage subclass for errors returned by data checks. |
| *DataCheckMessage* | Base class for a message returned by a DataCheck, tagged by name. |
| *DataCheckMessageCode* | Enum for data check message code. |
| *DataCheckMessageType* | Enum for type of data check message: WARNING or ERROR. |
| *DataChecks* | A collection of data checks. |
| *DataCheckWarning* | DataCheckMessage subclass for warnings returned by data checks. |
| *DateTimeFormatDataCheck* | Check if the datetime column has equally spaced intervals and is monotonically increasing or decreasing in order to be supported by time series estimators. |
| *DCAOParameterAllowedValuesType* | Enum for data check action option parameter allowed values type. |
| *DCAOParameterType* | Enum for data check action option parameter type. |
| *DefaultDataChecks* | A collection of basic data checks that is used by AutoML by default. |
| *IDColumnsDataCheck* | Check if any of the features are likely to be ID columns. |
| *InvalidTargetDataCheck* | Check if the target data is considered invalid. |
| *MulticollinearityDataCheck* | Check if any set features are likely to be multicollinear. |
| *NoVarianceDataCheck* | Check if the target or any of the features have no variance. |
| *NullDataCheck* | Check if there are any highly-null numerical, boolean, categorical, natural language, and unknown columns and rows in the input. |
| *OutliersDataCheck* | Checks if there are any outliers in input data by using IQR to determine score anomalies. |
| *SparsityDataCheck* | Check if there are any columns with sparsely populated values in the input. |
| *TargetDistributionDataCheck* | Check if the target data contains certain distributions that may need to be transformed prior training to improve model performance. Uses the Shapiro-Wilks test when the dataset is <=5000 samples, otherwise uses Jarque-Bera. |
| *TargetLeakageDataCheck* | Check if any of the features are highly correlated with the target by using mutual information, Pearson correlation, and other correlation metrics. |
| *TimeSeriesParametersDataCheck* | Checks whether the time series parameters are compatible with data splitting. |
| *TimeSeriesSplittingDataCheck* | Checks whether the time series target data is compatible with splitting. |
| *UniquenessDataCheck* | Check if there are any columns in the input that are either too unique for classification problems or not unique enough for regression problems. |

### Contents

**class** evalml.data_checks.**ClassImbalanceDataCheck**(*threshold=0.1*, *min_samples=100*, *num_cv_folds=3*, *test_size=None*)

Check if any of the target labels are imbalanced, or if the number of values for each target are below 2 times the number of CV folds. Use for classification problems.

**Parameters**

- **threshold** (*float*) – The minimum threshold allowed for class imbalance before a warning is raised. This threshold is calculated by comparing the number of samples in each class to the sum of samples in that class and the majority class. For example, a multiclass case with [900, 900, 100] samples per classes 0, 1, and 2, respectively, would have a 0.10 threshold for class 2 (100 / (900 + 100)). Defaults to 0.10.
- **min_samples** (*int*) – The minimum number of samples per accepted class. If the minority class is both below the threshold and min_samples, then we consider this severely imbalanced. Must be greater than 0. Defaults to 100.
- **num_cv_folds** (*int*) – The number of cross-validation folds. Must be positive. Choose 0 to ignore this warning. Defaults to 3.
- **test_size** (*None, float, int*) – Percentage of test set size. Used to calculate class imbalance prior to splitting the data into training and validation/test sets.

**Raises**

- **ValueError** – If threshold is not within 0 and 0.5
- **ValueError** – If min_samples is not greater than 0
- **ValueError** – If number of cv folds is negative
- **ValueError** – If test_size is not between 0 and 1

**Methods**

| | |
|---|---|
| *name* | Return a name describing the data check. |
| *validate* | Check if any target labels are imbalanced beyond a threshold for binary and multiclass problems. |

**name**(*cls*)

Return a name describing the data check.

**validate**(*self*, *X*, *y*)

Check if any target labels are imbalanced beyond a threshold for binary and multiclass problems.

Ignores NaN values in target labels if they appear.

**Parameters**

- **X** (*pd.DataFrame, np.ndarray*) – Features. Ignored.
- **y** (*pd.Series, np.ndarray*) – Target labels to check for imbalanced data.

**Returns**

**Dictionary with DataCheckWarnings if imbalance in classes is less than the threshold,** and DataCheckErrors if the number of values for each target is below 2 * num_cv_folds.

**Return type** dict

### Examples

```
>>> import pandas as pd
...
>>> X = pd.DataFrame()
>>> y = pd.Series([0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1])
```

In this binary example, the target class 0 is present in fewer than 10% (threshold=0.10) of instances, and fewer than 2 * the number of cross folds (2 * 3 = 6). Therefore, both a warning and an error are returned as part of the Class Imbalance Data Check. In addition, if a target is present with fewer than *min_samples* occurrences (default is 100) and is under the threshold, a severe class imbalance warning will be raised.

```
>>> class_imb_dc = ClassImbalanceDataCheck(threshold=0.10)
>>> assert class_imb_dc.validate(X, y) == [
...     {
...         "message": "The number of instances of these targets is less than 2␣
↪* the number of cross folds = 6 instances: [0]",
...         "data_check_name": "ClassImbalanceDataCheck",
...         "level": "error",
...         "code": "CLASS_IMBALANCE_BELOW_FOLDS",
...         "details": {"target_values": [0], "rows": None, "columns": None},
...         "action_options": []
...     },
...     {
...         "message": "The following labels fall below 10% of the target: [0]",
...         "data_check_name": "ClassImbalanceDataCheck",
...         "level": "warning",
...         "code": "CLASS_IMBALANCE_BELOW_THRESHOLD",
...         "details": {"target_values": [0], "rows": None, "columns": None},
...         "action_options": []
...     },
...     {
...         "message": "The following labels in the target have severe class␣
↪imbalance because they fall under 10% of the target and have less than 100␣
↪samples: [0]",
...         "data_check_name": "ClassImbalanceDataCheck",
...         "level": "warning",
...         "code": "CLASS_IMBALANCE_SEVERE",
...         "details": {"target_values": [0], "rows": None, "columns": None},
...         "action_options": []
...     }
... ]
```

In this multiclass example, the target class 0 is present in fewer than 30% of observations, however with 1 cv fold, the minimum number of instances required is 2 * 1 = 2. Therefore a warning, but not an error, is raised.

```
>>> y = pd.Series([0, 0, 1, 1, 1, 1, 2, 2, 2, 2, 2])
>>> class_imb_dc = ClassImbalanceDataCheck(threshold=0.30, min_samples=5, num_
↪cv_folds=1)
>>> assert class_imb_dc.validate(X, y) == [
...     {
...         "message": "The following labels fall below 30% of the target: [0]",
```

(continues on next page)

(continued from previous page)

```
...         "data_check_name": "ClassImbalanceDataCheck",
...         "level": "warning",
...         "code": "CLASS_IMBALANCE_BELOW_THRESHOLD",
...         "details": {"target_values": [0], "rows": None, "columns": None},
...         "action_options": []
...     },
...     {
...         "message": "The following labels in the target have severe class␣
↪imbalance because they fall under 30% of the target and have less than 5␣
↪samples: [0]",
...         "data_check_name": "ClassImbalanceDataCheck",
...         "level": "warning",
...         "code": "CLASS_IMBALANCE_SEVERE",
...         "details": {"target_values": [0], "rows": None, "columns": None},
...         "action_options": []
...     }
... ]
...
>>> y = pd.Series([0, 0, 1, 1, 1, 1, 2, 2, 2, 2])
>>> class_imb_dc = ClassImbalanceDataCheck(threshold=0.30, num_cv_folds=1)
>>> assert class_imb_dc.validate(X, y) == []
```

**class** evalml.data_checks.**DataCheck**

Base class for all data checks.

Data checks are a set of heuristics used to determine if there are problems with input data.

**Methods**

| | |
|---|---|
| *name* | Return a name describing the data check. |
| *validate* | Inspect and validate the input data, runs any necessary calculations or algorithms, and returns a list of warnings and errors if applicable. |

**name**(*cls*)

Return a name describing the data check.

**abstract validate**(*self*, *X*, *y=None*)

Inspect and validate the input data, runs any necessary calculations or algorithms, and returns a list of warnings and errors if applicable.

**Parameters**

- **X** (*pd.DataFrame*) – The input data of shape [n_samples, n_features]
- **y** (*pd.Series, optional*) – The target data of length [n_samples]

**Returns** Dictionary of DataCheckError and DataCheckWarning messages

**Return type** dict (*DataCheckMessage*)

**class** evalml.data_checks.**DataCheckAction**(*action_code*, *data_check_name*, *metadata=None*)

A recommended action returned by a DataCheck.

**Parameters**

- **action_code** (*str, DataCheckActionCode*) – Action code associated with the action.

- **data_check_name** (*str*) – Name of data check.
- **metadata** (*dict, optional*) – Additional useful information associated with the action. Defaults to None.

**Methods**

| | |
|---|---|
| *convert_dict_to_action* | Convert a dictionary into a DataCheckAction. |
| *to_dict* | Return a dictionary form of the data check action. |

**static convert_dict_to_action**(*action_dict*)

Convert a dictionary into a DataCheckAction.

**Parameters** **action_dict** – Dictionary to convert into action. Should have keys "code", "data_check_name", and "metadata".

**Raises** **ValueError** – If input dictionary does not have keys *code* and *metadata* and if the *metadata* dictionary does not have keys *columns* and *rows*.

**Returns** DataCheckAction object from the input dictionary.

**to_dict**(*self*)

Return a dictionary form of the data check action.

**class** evalml.data_checks.**DataCheckActionCode**

Enum for data check action code.

**Attributes**

| | |
|---|---|
| **DROP_COL** | Action code for dropping a column. |
| **DROP_ROWS** | Action code for dropping rows. |
| **IMPUTE_COL** | Action code for imputing a column. |
| **REGULARIZE_AND_IMPUTE_DATASET** | Action code for regularizing and imputing all features and target time series data. |
| **SET_FIRST_COL_ID** | Action code for setting the first column as an id column. |
| **TRANSFORM_TARGET** | Action code for transforming the target data. |

**Methods**

| | |
|---|---|
| *name* | The name of the Enum member. |
| *value* | The value of the Enum member. |

**name**(*self*)

The name of the Enum member.

**value**(*self*)

The value of the Enum member.

**class** evalml.data_checks.**DataCheckActionOption**(*action_code*, *data_check_name*, *parameters=None*, *metadata=None*)

A recommended action option returned by a DataCheck.

It contains an action code that indicates what the action should be, a data check name that indicates what data check was used to generate the action, and parameters and metadata which can be used to further refine the action.

**Parameters**

- **action_code** (DataCheckActionCode) – Action code associated with the action option.
- **data_check_name** (*str*) – Name of the data check that produced this option.
- **parameters** (*dict*) – Parameters associated with the action option. Defaults to None.
- **metadata** (*dict, optional*) – Additional useful information associated with the action option. Defaults to None.

### Examples

```
>>> parameters = {
...     "global_parameter_name": {
...         "parameter_type": "global",
...         "type": "float",
...         "default_value": 0.0,
...     },
...     "column_parameter_name": {
...         "parameter_type": "column",
...         "columns": {
...             "a": {
...                 "impute_strategy": {
...                     "categories": ["mean", "most_frequent"],
...                     "type": "category",
...                     "default_value": "mean",
...                 },
...             "constant_fill_value": {"type": "float", "default_value": 0},
...             },
...         },
...     },
... }
>>> data_check_action = DataCheckActionOption(DataCheckActionCode.DROP_COL, None,␣
↪metadata={}, parameters=parameters)
```

**Methods**

| | |
|---|---|
| *convert_dict_to_option* | Convert a dictionary into a DataCheckActionOption. |
| *get_action_from_defaults* | Returns an action based on the defaults parameters. |
| *to_dict* | Return a dictionary form of the data check action option. |

**static convert_dict_to_option**(*action_dict*)

Convert a dictionary into a DataCheckActionOption.

**Parameters** **action_dict** – Dictionary to convert into an action option. Should have keys "code", "data_check_name", and "metadata".

**Raises** **ValueError** – If input dictionary does not have keys *code* and *metadata* and if the *metadata* dictionary does not have keys *columns* and *rows*.

**Returns** DataCheckActionOption object from the input dictionary.

**get_action_from_defaults**(*self*)

Returns an action based on the defaults parameters.

**Returns** An based on the defaults parameters the option.

**Return type** *DataCheckAction*

`to_dict`(*self*)

Return a dictionary form of the data check action option.

**class** `evalml.data_checks.`**`DataCheckError`**(*message*, *data_check_name*, *message_code=None*, *details=None*, *action_options=None*)

DataCheckMessage subclass for errors returned by data checks.

**Attributes**

| | |
|---|---|
| **message_type** | DataCheckMessageType.ERROR |

**Methods**

| | |
|---|---|
| *to_dict* | Return a dictionary form of the data check message. |

`to_dict`(*self*)

Return a dictionary form of the data check message.

**class** `evalml.data_checks.`**`DataCheckMessage`**(*message*, *data_check_name*, *message_code=None*, *details=None*, *action_options=None*)

Base class for a message returned by a DataCheck, tagged by name.

**Parameters**

- **message** (*str*) – Message string.
- **data_check_name** (*str*) – Name of the associated data check.
- **message_code** (*DataCheckMessageCode, optional*) – Message code associated with the message. Defaults to None.
- **details** (*dict, optional*) – Additional useful information associated with the message. Defaults to None.
- **action_options** (*list, optional*) – A list of `DataCheckActionOption`s associated with the message. Defaults to None.

**Attributes**

| | |
|---|---|
| **message_type** | None |

**Methods**

| | |
|---|---|
| *to_dict* | Return a dictionary form of the data check message. |

`to_dict`(*self*)

Return a dictionary form of the data check message.

**class** evalml.data_checks.**DataCheckMessageCode**

Enum for data check message code.

**Attributes**

| | |
|---|---|
| **CLASS_IMBALANCE_BELOW_FOLDS** | Message code for when the number of values for each target is below 2 * number of CV folds. |
| **CLASS_IMBALANCE_BELOW_THRESHOLD** | Message code for when imbalance in classes is less than the threshold. |
| **CLASS_IMBALANCE_SEVERE** | Message code for when balance in classes is less than the threshold and minimum class is less than minimum number of accepted samples. |
| **COLS_WITH_NULL** | Message code for columns with null values. |
| **DATETIME_HAS_MISALIGNED_VALUES** | Message code for when datetime information has values that are not aligned with the inferred frequency. |
| **DATETIME_HAS_NAN** | Message code for when input datetime columns contain NaN values. |
| **DATETIME_HAS_REDUNDANT_ROW** | Message code for when datetime information has more than one row per datetime. |
| **DATETIME_HAS_UNEVEN_INTERVALS** | Message code for when the datetime values have uneven intervals. |
| **DATETIME_INFORMATION_NOT_FOUND** | Message code for when datetime information can not be found or is in an unaccepted format. |
| **DATETIME_IS_MISSING_VALUES** | Message code for when datetime feature has values missing between the start and end dates. |
| **DATETIME_IS_NOT_MONOTONIC** | Message code for when the datetime values are not monotonically increasing. |
| **DATETIME_NO_FREQUENCY_INFERRED** | Message code for when no frequency can be inferred in the datetime values through Woodwork. |
| **HAS_ID_COLUMN** | Message code for data that has ID columns. |
| **HAS_ID_FIRST_COLUMN** | Message code for data that has an ID column as the first column. |
| **HAS_OUTLIERS** | Message code for when outliers are detected. |
| **HIGH_VARIANCE** | Message code for when high variance is detected for cross-validation. |
| **HIGHLY_NULL_COLS** | Message code for highly null columns. |
| **HIGHLY_NULL_ROWS** | Message code for highly null rows. |
| **IS_MULTICOLLINEAR** | Message code for when data is potentially multicollinear. |
| **MISMATCHED_INDICES** | Message code for when input target and features have mismatched indices. |
| **MISMATCHED_INDICES_ORDER** | Message code for when input target and features have mismatched indices order. The two inputs have the same index values, but shuffled. |
| **MISMATCHED_LENGTHS** | Message code for when input target and features have different lengths. |
| **NATURAL_LANGUAGE_HAS_NAN** | Message code for when input natural language columns contain NaN values. |
| **NO_VARIANCE** | Message code for when data has no variance (1 unique value). |
| **NO_VARIANCE_WITH_NULL** | Message code for when data has one unique value and NaN values. |
| **NO_VARIANCE_ZERO_UNIQUE** | Message code for when data has no variance (0 unique value) |
| **NOT_UNIQUE_ENOUGH** | Message code for when data does not possess enough unique values. |
| **TARGET_BINARY_NOT_TWO_UNIQUE_VALUES** | Message code for target data for a binary classification problem that does not have two unique values. |
| **TARGET_HAS_NULL** | Message code for target data that has null values. |
| **TARGET_INCOMPATIBLE_OBJECTIVE** | Message code for target data that has incompatible values for the specified objective |

continues on next page

Table 2 – continued from previous page

| | |
|---|---|
| **TARGET_IS_EMPTY_OR_FULLY_NULL** | Message code for target data that is empty or has all null values. |
| **TARGET_IS_NONE** | Message code for when target is None. |
| **TARGET_LEAKAGE** | Message code for when target leakage is detected. |
| **TARGET_LOGNORMAL_DISTRIBUTION** | Message code for target data with a lognormal distribution. |
| **TARGET_MULTICLASS_HIGH_UNIQUE_CLASS** | Message code for target data for a multi classification problem that has an abnormally large number of unique classes relative to the number of target values. |
| **TARGET_MULTICLASS_NOT_ENOUGH_CLASSES** | Message code for target data for a multi classification problem that does not have more than two unique classes. |
| **TARGET_MULTICLASS_NOT_TWO_EXAMPLES_PER_CLASS** | Message code for target data for a multi classification problem that does not have two examples per class. |
| **TARGET_UNSUPPORTED_PROBLEM_TYPE** | Message code for target data that is being checked against an unsupported problem type. |
| **TARGET_UNSUPPORTED_TYPE** | Message code for target data that is of an unsupported type. |
| **TARGET_UNSUPPORTED_TYPE_REGRESSION** | Message code for target data that is incompatible with regression |
| **TIME_SERIES_PARAMETERS_NOT_COMPATIBLE_WITH_SPLIT** | Message code when the time series parameters are too large for the smallest data split. |
| **TIME_SERIES_TARGET_NOT_COMPATIBLE_WITH_SPLIT** | Message code when any training and validation split of the time series target doesn't contain all classes. |
| **TOO_SPARSE** | Message code for when multiclass data has values that are too sparsely populated. |
| **TOO_UNIQUE** | Message code for when data possesses too many unique values. |

**Methods**

| | |
|---|---|
| *name* | The name of the Enum member. |
| *value* | The value of the Enum member. |

**name**(*self*)

The name of the Enum member.

**value**(*self*)

The value of the Enum member.

**class** evalml.data_checks.**DataCheckMessageType**

Enum for type of data check message: WARNING or ERROR.

**Attributes**

| | |
|---|---|
| **ERROR** | Error message returned by a data check. |
| **WARNING** | Warning message returned by a data check. |

**Methods**

| | |
|---|---|
| *name* | The name of the Enum member. |
| *value* | The value of the Enum member. |

**name**(*self*)

The name of the Enum member.

**value**(*self*)

The value of the Enum member.

**class** evalml.data_checks.**DataChecks**(*data_checks=None*, *data_check_params=None*)

A collection of data checks.

**Parameters**

- **data_checks** (*list (DataCheck)*) – List of DataCheck objects.
- **data_check_params** (*dict*) – Parameters for passed DataCheck objects.

**Methods**

| | |
|---|---|
| *validate* | Inspect and validate the input data against data checks and returns a list of warnings and errors if applicable. |

**validate**(*self*, *X*, *y=None*)

Inspect and validate the input data against data checks and returns a list of warnings and errors if applicable.

**Parameters**

- **X** (*pd.DataFrame, np.ndarray*) – The input data of shape [n_samples, n_features]
- **y** (*pd.Series, np.ndarray*) – The target data of length [n_samples]

**Returns** Dictionary containing DataCheckMessage objects

**Return type** dict

**class** evalml.data_checks.**DataCheckWarning**(*message*, *data_check_name*, *message_code=None*, *details=None*, *action_options=None*)

DataCheckMessage subclass for warnings returned by data checks.

**Attributes**

| | |
|---|---|
| **message_type** | DataCheckMessageType.WARNING |

**Methods**

| | |
|---|---|
| *to_dict* | Return a dictionary form of the data check message. |

**to_dict**(*self*)

Return a dictionary form of the data check message.

**class** evalml.data_checks.**DateTimeFormatDataCheck**(*datetime_column='index'*, *nan_duplicate_threshold=0.75*)

Check if the datetime column has equally spaced intervals and is monotonically increasing or decreasing in order to be supported by time series estimators.

**Parameters**

- **datetime_column** (*str, int*) – The name of the datetime column. If the datetime values are in the index, then pass "index".

- **nan_duplicate_threshold** (*float*) – The percentage of values in the *datetime_column* that must not be duplicate or nan before *DATETIME_NO_FREQUENCY_INFERRED* is returned instead of *DATETIME_HAS_UNEVEN_INTERVALS*. For example, if this is set to 0.80, then only 20% of the values in *datetime_column* can be duplicate or nan. Defaults to 0.75.

**Methods**

| | |
|---|---|
| *name* | Return a name describing the data check. |
| *validate* | Checks if the target data has equal intervals and is monotonically increasing. |

**name**(*cls*)

Return a name describing the data check.

**validate**(*self*, *X*, *y*)

Checks if the target data has equal intervals and is monotonically increasing.

Will return a DataCheckError if the data is not a datetime type, is not increasing, has redundant or missing row(s), contains invalid (NaN or None) values, or has values that don't align with the assumed frequency.

**Parameters**

- **X** (*pd.DataFrame, np.ndarray*) – Features.
- **y** (*pd.Series, np.ndarray*) – Target data.

**Returns** List with DataCheckErrors if unequal intervals are found in the datetime column.

**Return type** dict (*DataCheckError*)

### Examples

```
>>> import pandas as pd
```

The column 'dates' has a set of two dates with daily frequency, two dates with hourly frequency, and two dates with monthly frequency.

```
>>> X = pd.DataFrame(pd.date_range("2015-01-01", periods=2).append(pd.date_
↪range("2015-01-08", periods=2, freq="H").append(pd.date_range("2016-03-02",␣
↪periods=2, freq="M"))), columns=["dates"])
>>> y = pd.Series([0, 1, 0, 1, 1, 0])
>>> datetime_format_dc = DateTimeFormatDataCheck(datetime_column="dates")
>>> assert datetime_format_dc.validate(X, y) == [
...     {
...         "message": "No frequency could be detected in column 'dates',␣
↪possibly due to uneven intervals or too many duplicate/missing values.",
...         "data_check_name": "DateTimeFormatDataCheck",
...         "level": "error",
...         "code": "DATETIME_NO_FREQUENCY_INFERRED",
...         "details": {"columns": None, "rows": None},
...         "action_options": []
...     }
... ]
```

The column "dates" has a gap in the values, which implies there are many dates missing.

```
>>> X = pd.DataFrame(pd.date_range("2021-01-01", periods=9).append(pd.date_
↪range("2021-01-31", periods=50)), columns=["dates"])
>>> y = pd.Series([0, 1, 0, 1, 1, 0, 0, 0, 1, 0])
>>> ww_payload = infer_frequency(X["dates"], debug=True, window_length=5,␣
↪threshold=0.8)
>>> datetime_format_dc = DateTimeFormatDataCheck(datetime_column="dates")
>>> assert datetime_format_dc.validate(X, y) == [
...     {
...         "message": "Column 'dates' has datetime values missing between␣
↪start and end date.",
...         "data_check_name": "DateTimeFormatDataCheck",
...         "level": "error",
...         "code": "DATETIME_IS_MISSING_VALUES",
...         "details": {"columns": None, "rows": None},
...         "action_options": []
...      },
...     {
...         "message": "A frequency was detected in column 'dates', but there␣
↪are faulty datetime values that need to be addressed.",
...         "data_check_name": "DateTimeFormatDataCheck",
...         "level": "error",
...         "code": "DATETIME_HAS_UNEVEN_INTERVALS",
...         "details": {'columns': None, 'rows': None},
...         "action_options": [
...             {
...                 'code': 'REGULARIZE_AND_IMPUTE_DATASET',
...                 'data_check_name': 'DateTimeFormatDataCheck',
...                 'metadata': {
...                         'columns': None,
...                         'is_target': True,
...                         'rows': None
...                 },
...                 'parameters': {
...                         'time_index': {
...                             'default_value': 'dates',
...                             'parameter_type': 'global',
...                             'type': 'str'
...                         },
...                         'frequency_payload': {
...                             'default_value': ww_payload,
...                             'parameter_type': 'global',
...                             'type': 'tuple'
...                         }
...                 }
...             }
...         ]
...     }
... ]
```

The column "dates" has a repeat of the date 2021-01-09 appended to the end, which is considered redundant and will raise an error.

```
>>> X = pd.DataFrame(pd.date_range("2021-01-01", periods=9).append(pd.date_
↪range("2021-01-09", periods=1)), columns=["dates"])
```

(continues on next page)

(continued from previous page)

```
>>> y = pd.Series([0, 1, 0, 1, 1, 0, 0, 0, 1, 0])
>>> ww_payload = infer_frequency(X["dates"], debug=True, window_length=5,␣
↪threshold=0.8)
>>> datetime_format_dc = DateTimeFormatDataCheck(datetime_column="dates")
>>> assert datetime_format_dc.validate(X, y) == [
...     {
...         "message": "Column 'dates' has more than one row with the same␣
↪datetime value.",
...         "data_check_name": "DateTimeFormatDataCheck",
...         "level": "error",
...         "code": "DATETIME_HAS_REDUNDANT_ROW",
...         "details": {"columns": None, "rows": None},
...         "action_options": []
...      },
...     {
...         "message": "A frequency was detected in column 'dates', but there␣
↪are faulty datetime values that need to be addressed.",
...         "data_check_name": "DateTimeFormatDataCheck",
...         "level": "error",
...         "code": "DATETIME_HAS_UNEVEN_INTERVALS",
...         "details": {'columns': None, 'rows': None},
...         "action_options": [
...             {
...                 'code': 'REGULARIZE_AND_IMPUTE_DATASET',
...                 'data_check_name': 'DateTimeFormatDataCheck',
...                 'metadata': {
...                         'columns': None,
...                         'is_target': True,
...                         'rows': None
...                 },
...                 'parameters': {
...                         'time_index': {
...                             'default_value': 'dates',
...                             'parameter_type': 'global',
...                             'type': 'str'
...                         },
...                         'frequency_payload': {
...                             'default_value': ww_payload,
...                             'parameter_type': 'global',
...                             'type': 'tuple'
...                         }
...                 }
...             }
...         ]
...     }
... ]
```

The column "Weeks" has a date that does not follow the weekly pattern, which is considered misaligned.

```
>>> X = pd.DataFrame(pd.date_range("2021-01-01", freq="W", periods=12).
↪append(pd.date_range("2021-03-22", periods=1)), columns=["Weeks"])
>>> ww_payload = infer_frequency(X["Weeks"], debug=True, window_length=5,␣
↪threshold=0.8)
```

(continues on next page)

(continued from previous page)

```
>>> datetime_format_dc = DateTimeFormatDataCheck(datetime_column="Weeks")
>>> assert datetime_format_dc.validate(X, y) == [
...     {
...         "message": "Column 'Weeks' has datetime values that do not align␣
↪with the inferred frequency.",
...         "data_check_name": "DateTimeFormatDataCheck",
...         "level": "error",
...         "details": {"columns": None, "rows": None},
...         "code": "DATETIME_HAS_MISALIGNED_VALUES",
...         "action_options": []
...      },
...     {
...         "message": "A frequency was detected in column 'Weeks', but there␣
↪are faulty datetime values that need to be addressed.",
...         "data_check_name": "DateTimeFormatDataCheck",
...         "level": "error",
...         "code": "DATETIME_HAS_UNEVEN_INTERVALS",
...         "details": {'columns': None, 'rows': None},
...         "action_options": [
...             {
...                 'code': 'REGULARIZE_AND_IMPUTE_DATASET',
...                 'data_check_name': 'DateTimeFormatDataCheck',
...                 'metadata': {
...                         'columns': None,
...                         'is_target': True,
...                         'rows': None
...                 },
...                 'parameters': {
...                         'time_index': {
...                             'default_value': 'Weeks',
...                             'parameter_type': 'global',
...                             'type': 'str'
...                         },
...                         'frequency_payload': {
...                             'default_value': ww_payload,
...                             'parameter_type': 'global',
...                             'type': 'tuple'
...                         }
...                 }
...             }
...         ]
...     }
... ]
```

The column "Weeks" has a date that does not follow the weekly pattern, which is considered misaligned.

```
>>> X = pd.DataFrame(pd.date_range("2021-01-01", freq="W", periods=12).
↪append(pd.date_range("2021-03-22", periods=1)), columns=["Weeks"])
>>> ww_payload = infer_frequency(X["Weeks"], debug=True, window_length=5,␣
↪threshold=0.8)
>>> datetime_format_dc = DateTimeFormatDataCheck(datetime_column="Weeks")
>>> assert datetime_format_dc.validate(X, y) == [
```

(continues on next page)

(continued from previous page)

```
...     {
...         "message": "Column 'Weeks' has datetime values that do not align␣
↪with the inferred frequency.",
...         "data_check_name": "DateTimeFormatDataCheck",
...         "level": "error",
...         "details": {"columns": None, "rows": None},
...         "code": "DATETIME_HAS_MISALIGNED_VALUES",
...         "action_options": []
...      },
...     {
...         "message": "A frequency was detected in column 'Weeks', but there␣
↪are faulty datetime values that need to be addressed.",
...         "data_check_name": "DateTimeFormatDataCheck",
...         "level": "error",
...         "code": "DATETIME_HAS_UNEVEN_INTERVALS",
...         "details": {'columns': None, 'rows': None},
...         "action_options": [
...             {
...                 'code': 'REGULARIZE_AND_IMPUTE_DATASET',
...                 'data_check_name': 'DateTimeFormatDataCheck',
...                 'metadata': {
...                         'columns': None,
...                         'is_target': True,
...                         'rows': None
...                 },
...                 'parameters': {
...                         'time_index': {
...                             'default_value': 'Weeks',
...                             'parameter_type': 'global',
...                             'type': 'str'
...                         },
...                         'frequency_payload': {
...                             'default_value': ww_payload,
...                             'parameter_type': 'global',
...                             'type': 'tuple'
...                         }
...                 }
...             }
...         ]
...     }
... ]
```

The column "Weeks" passed integers instead of datetime data, which will raise an error.

```
>>> X = pd.DataFrame([1, 2, 3, 4], columns=["Weeks"])
>>> y = pd.Series([0] * 4)
>>> datetime_format_dc = DateTimeFormatDataCheck(datetime_column="Weeks")
>>> assert datetime_format_dc.validate(X, y) == [
...     {
...         "message": "Datetime information could not be found in the data, or␣
↪was not in a supported datetime format.",
...         "data_check_name": "DateTimeFormatDataCheck",
```

(continues on next page)

(continued from previous page)

```
...         "level": "error",
...         "details": {"columns": None, "rows": None},
...         "code": "DATETIME_INFORMATION_NOT_FOUND",
...         "action_options": []
...      }
... ]
```

Converting that same integer data to datetime, however, is valid.

```
>>> X = pd.DataFrame(pd.to_datetime([1, 2, 3, 4]), columns=["Weeks"])
>>> datetime_format_dc = DateTimeFormatDataCheck(datetime_column="Weeks")
>>> assert datetime_format_dc.validate(X, y) == []
```

```
>>> X = pd.DataFrame(pd.date_range("2021-01-01", freq="W", periods=10),␣
↪columns=["Weeks"])
>>> datetime_format_dc = DateTimeFormatDataCheck(datetime_column="Weeks")
>>> assert datetime_format_dc.validate(X, y) == []
```

While the data passed in is of datetime type, time series requires the datetime information in datetime_column to be monotonically increasing (ascending).

```
>>> X = X.iloc[::-1]
>>> datetime_format_dc = DateTimeFormatDataCheck(datetime_column="Weeks")
>>> assert datetime_format_dc.validate(X, y) == [
...     {
...         "message": "Datetime values must be sorted in ascending order.",
...         "data_check_name": "DateTimeFormatDataCheck",
...         "level": "error",
...         "details": {"columns": None, "rows": None},
...         "code": "DATETIME_IS_NOT_MONOTONIC",
...         "action_options": []
...      }
... ]
```

The first value in the column "index" is replaced with NaT, which will raise an error in this data check.

```
>>> dates = [["2-1-21", "3-1-21"],
...         ["2-2-21", "3-2-21"],
...         ["2-3-21", "3-3-21"],
...         ["2-4-21", "3-4-21"],
...         ["2-5-21", "3-5-21"],
...         ["2-6-21", "3-6-21"],
...         ["2-7-21", "3-7-21"],
...         ["2-8-21", "3-8-21"],
...         ["2-9-21", "3-9-21"],
...         ["2-10-21", "3-10-21"],
...         ["2-11-21", "3-11-21"],
...         ["2-12-21", "3-12-21"]]
>>> dates[0][0] = None
>>> df = pd.DataFrame(dates, columns=["days", "days2"])
>>> ww_payload = infer_frequency(pd.to_datetime(df["days"]), debug=True, window_
↪length=5, threshold=0.8)
```

(continues on next page)

(continued from previous page)

```
>>> datetime_format_dc = DateTimeFormatDataCheck(datetime_column="days")
>>> assert datetime_format_dc.validate(df, y) == [
...     {
...         "message": "Input datetime column 'days' contains NaN values.␣
↪Please impute NaN values or drop these rows.",
...         "data_check_name": "DateTimeFormatDataCheck",
...         "level": "error",
...         "details": {"columns": None, "rows": None},
...         "code": "DATETIME_HAS_NAN",
...         "action_options": []
...      },
...     {
...         "message": "A frequency was detected in column 'days', but there␣
↪are faulty datetime values that need to be addressed.",
...         "data_check_name": "DateTimeFormatDataCheck",
...         "level": "error",
...         "code": "DATETIME_HAS_UNEVEN_INTERVALS",
...         "details": {'columns': None, 'rows': None},
...         "action_options": [
...             {
...                 'code': 'REGULARIZE_AND_IMPUTE_DATASET',
...                 'data_check_name': 'DateTimeFormatDataCheck',
...                 'metadata': {
...                         'columns': None,
...                         'is_target': True,
...                         'rows': None
...                 },
...                 'parameters': {
...                         'time_index': {
...                             'default_value': 'days',
...                             'parameter_type': 'global',
...                             'type': 'str'
...                         },
...                         'frequency_payload': {
...                             'default_value': ww_payload,
...                             'parameter_type': 'global',
...                             'type': 'tuple'
...                         }
...                 }
...             }
...         ]
...     }
... ]
...
```

**class** evalml.data_checks.**DCAOParameterAllowedValuesType**

Enum for data check action option parameter allowed values type.

**Attributes**

| | |
|---|---|
| **CATEGORICAL** | Categorical allowed values type. Parameters that have a set of allowed values. |
| **NUMERICAL** | Numerical allowed values type. Parameters that have a range of allowed values. |

**Methods**

| | |
|---|---|
| *name* | The name of the Enum member. |
| *value* | The value of the Enum member. |

**name**(*self*)

The name of the Enum member.

**value**(*self*)

The value of the Enum member.

**class** evalml.data_checks.**DCAOParameterType**

Enum for data check action option parameter type.

**Attributes**

| | |
|---|---|
| **COLUMN** | Column parameter type. Parameters that apply to a specific column in the data set. |
| **GLOBAL** | Global parameter type. Parameters that apply to the entire data set. |

**Methods**

| | |
|---|---|
| *all_parameter_types* | Get a list of all defined parameter types. |
| *handle_dcao_parameter_type* | Handles the data check action option parameter type by either returning the DCAOParameterType enum or converting from a str. |
| *name* | The name of the Enum member. |
| *value* | The value of the Enum member. |

**all_parameter_types**(*cls*)

Get a list of all defined parameter types.

**Returns** List of all defined parameter types.

**Return type** list(*DCAOParameterType*)

**static handle_dcao_parameter_type**(*dcao_parameter_type*)

Handles the data check action option parameter type by either returning the DCAOParameterType enum or converting from a str.

**Parameters** **dcao_parameter_type** (*str or* DCAOParameterType) – Data check action option parameter type that needs to be handled.

**Returns** DCAOParameterType enum

**Raises**

- **KeyError** – If input is not a valid DCAOParameterType enum value.
- **ValueError** – If input is not a string or DCAOParameterType object.

**name**(*self*)

The name of the Enum member.

**value**(*self*)

The value of the Enum member.

**class** evalml.data_checks.**DefaultDataChecks**(*problem_type*, *objective*, *n_splits=3*, *problem_configuration=None*)

A collection of basic data checks that is used by AutoML by default.

Includes:

- *NullDataCheck*
- *HighlyNullRowsDataCheck*
- *IDColumnsDataCheck*
- *TargetLeakageDataCheck*
- *InvalidTargetDataCheck*
- *NoVarianceDataCheck*
- *ClassImbalanceDataCheck* (for classification problem types)
- *TargetDistributionDataCheck* (for regression problem types)
- *DateTimeFormatDataCheck* (for time series problem types)
- 'TimeSeriesParametersDataCheck' (for time series problem types)
- *TimeSeriesSplittingDataCheck* (for time series classification problem types)

**Parameters**

- **problem_type** (*str*) – The problem type that is being validated. Can be regression, binary, or multiclass.
- **objective** (*str or ObjectiveBase*) – Name or instance of the objective class.
- **n_splits** (*int*) – The number of splits as determined by the data splitter being used. Defaults to 3.
- **problem_configuration** (*dict*) – Required for time series problem types. Values should be passed in for time_index,
- **gap** –
- **forecast_horizon** –
- **max_delay.** (*and*) –

**Methods**

| | |
|---|---|
| *validate* | Inspect and validate the input data against data checks and returns a list of warnings and errors if applicable. |

**validate**(*self*, *X*, *y=None*)

Inspect and validate the input data against data checks and returns a list of warnings and errors if applicable.

**Parameters**

- **X** (*pd.DataFrame, np.ndarray*) – The input data of shape [n_samples, n_features]

- **y** (*pd.Series, np.ndarray*) – The target data of length [n_samples]

**Returns** Dictionary containing DataCheckMessage objects

**Return type** dict

**class** evalml.data_checks.**IDColumnsDataCheck**(*id_threshold=1.0*)

Check if any of the features are likely to be ID columns.

**Parameters** **id_threshold** (*float*) – The probability threshold to be considered an ID column. Defaults to 1.0.

**Methods**

| | |
|---|---|
| *name* | Return a name describing the data check. |
| *validate* | Check if any of the features are likely to be ID columns. Currently performs a number of simple checks. |

**name**(*cls*)

Return a name describing the data check.

**validate**(*self, X, y=None*)

Check if any of the features are likely to be ID columns. Currently performs a number of simple checks.

Checks performed are:

- column name is "id"
- column name ends in "_id"
- column contains all unique values (and is categorical / integer type)

**Parameters**

- **X** (*pd.DataFrame, np.ndarray*) – The input features to check.
- **y** (*pd.Series*) – The target. Defaults to None. Ignored.

**Returns** A dictionary of features with column name or index and their probability of being ID columns

**Return type** dict

### Examples

```
>>> import pandas as pd
```

Columns that end in "_id" and are completely unique are likely to be ID columns.

```
>>> df = pd.DataFrame({
...     "profits": [25, 15, 15, 31, 19],
...     "customer_id": [123, 124, 125, 126, 127],
...     "Sales": [10, 42, 31, 51, 61]
... })
...
>>> id_col_check = IDColumnsDataCheck()
>>> assert id_col_check.validate(df) == [
```

(continues on next page)

(continued from previous page)

```
...     {
...         "message": "Columns 'customer_id' are 100.0% or more likely to be␣
↪an ID column",
...         "data_check_name": "IDColumnsDataCheck",
...         "level": "warning",
...         "code": "HAS_ID_COLUMN",
...         "details": {"columns": ["customer_id"], "rows": None},
...         "action_options": [
...             {
...                 "code": "DROP_COL",
...                 "data_check_name": "IDColumnsDataCheck",
...                 "parameters": {},
...                 "metadata": {"columns": ["customer_id"], "rows": None}
...             }
...         ]
...     }
... ]
```

Columns named "ID" with all unique values will also be identified as ID columns.

```
>>> df = df.rename(columns={"customer_id": "ID"})
>>> id_col_check = IDColumnsDataCheck()
>>> assert id_col_check.validate(df) == [
...     {
...         "message": "Columns 'ID' are 100.0% or more likely to be an ID␣
↪column",
...         "data_check_name": "IDColumnsDataCheck",
...         "level": "warning",
...         "code": "HAS_ID_COLUMN",
...         "details": {"columns": ["ID"], "rows": None},
...         "action_options": [
...            {
...                 "code": "DROP_COL",
...                 "data_check_name": "IDColumnsDataCheck",
...                 "parameters": {},
...                 "metadata": {"columns": ["ID"], "rows": None}
...             }
...         ]
...     }
... ]
```

Despite being all unique, "Country_Rank" will not be identified as an ID column as id_threshold is set to 1.0 by default and its name doesn't indicate that it's an ID.

```
>>> df = pd.DataFrame({
...     "humidity": ["high", "very high", "low", "low", "high"],
...     "Country_Rank": [1, 2, 3, 4, 5],
...     "Sales": ["very high", "high", "high", "medium", "very low"]
... })
...
>>> id_col_check = IDColumnsDataCheck()
>>> assert id_col_check.validate(df) == []
```

However lowering the threshold will cause this column to be identified as an ID.

```
>>> id_col_check = IDColumnsDataCheck()
>>> id_col_check = IDColumnsDataCheck(id_threshold=0.95)
>>> assert id_col_check.validate(df) == [
...     {
...         "message": "Columns 'Country_Rank' are 95.0% or more likely to be␣
↪an ID column",
...         "data_check_name": "IDColumnsDataCheck",
...         "level": "warning",
...         "details": {"columns": ["Country_Rank"], "rows": None},
...         "code": "HAS_ID_COLUMN",
...         "action_options": [
...             {
...                 "code": "DROP_COL",
...                 "data_check_name": "IDColumnsDataCheck",
...                 "parameters": {},
...                 "metadata": {"columns": ["Country_Rank"], "rows": None}
...             }
...         ]
...     }
... ]
```

If the first column of the dataframe has all unique values and is named either 'ID' or a name that ends with '_id', it is probably the primary key. The other ID columns should be dropped.

```
>>> df = pd.DataFrame({
...     "sales_id": [0, 1, 2, 3, 4],
...     "customer_id": [123, 124, 125, 126, 127],
...     "Sales": [10, 42, 31, 51, 61]
... })
...
>>> id_col_check = IDColumnsDataCheck()
>>> assert id_col_check.validate(df) == [
...     {
...         "message": "The first column 'sales_id' is likely to be the primary␣
↪key",
...         "data_check_name": "IDColumnsDataCheck",
...         "level": "warning",
...         "code": "HAS_ID_FIRST_COLUMN",
...         "details": {"columns": ["sales_id"], "rows": None},
...         "action_options": [
...             {
...                 "code": "SET_FIRST_COL_ID",
...                 "data_check_name": "IDColumnsDataCheck",
...                 "parameters": {},
...                 "metadata": {"columns": ["sales_id"], "rows": None}
...             }
...         ]
...     },
...     {
...         "message": "Columns 'customer_id' are 100.0% or more likely to be an␣
↪ID column",
...         "data_check_name": "IDColumnsDataCheck",
```

(continues on next page)

(continued from previous page)

```
...         "level": "warning",
...         "code": "HAS_ID_COLUMN",
...         "details": {"columns": ["customer_id"], "rows": None},
...         "action_options": [
...             {
...                 "code": "DROP_COL",
...                 "data_check_name": "IDColumnsDataCheck",
...                 "parameters": {},
...                 "metadata": {"columns": ["customer_id"], "rows": None}
...             }
...         ]
...     }
... ]
```

**class** evalml.data_checks.**InvalidTargetDataCheck**(*problem_type*, *objective*, *n_unique=100*, *null_strategy='drop'*)

Check if the target data is considered invalid.

**Target data is considered invalid if:**

- Target is None.
- Target has NaN or None values.
- Target is of an unsupported Woodwork logical type.
- Target and features have different lengths or indices.
- Target does not have enough instances of a class in a classification problem.
- Target does not contain numeric data for regression problems.

**Parameters**

- **problem_type** (*str or ProblemTypes*) – The specific problem type to data check for. e.g. 'binary', 'multiclass', 'regression, 'time series regression'
- **objective** (*str or ObjectiveBase*) – Name or instance of the objective class.
- **n_unique** (*int*) – Number of unique target values to store when problem type is binary and target incorrectly has more than 2 unique values. Non-negative integer. If None, stores all unique values. Defaults to 100.
- **null_strategy** (*str*) – The type of action option that should be returned if the target is partially null. The options are *impute* and *drop* (default). *impute* - Will return a *DataCheckActionOption* for imputing the target column. *drop* - Will return a *DataCheckActionOption* for dropping the null rows in the target column.

**Attributes**

| | |
|---|---|
| **multi-class_continuous_threshold** | 0.05 |

**Methods**

| | |
|---|---|
| *name* | Return a name describing the data check. |
| *validate* | Check if the target data is considered invalid. If the input features argument is not None, it will be used to check that the target and features have the same dimensions and indices. |

**name**(*cls*)

Return a name describing the data check.

**validate**(*self*, *X*, *y*)

Check if the target data is considered invalid. If the input features argument is not None, it will be used to check that the target and features have the same dimensions and indices.

**Target data is considered invalid if:**

- Target is None.
- Target has NaN or None values.
- Target is of an unsupported Woodwork logical type.
- Target and features have different lengths or indices.
- Target does not have enough instances of a class in a classification problem.
- Target does not contain numeric data for regression problems.

**Parameters**

- **X** (*pd.DataFrame, np.ndarray*) – Features. If not None, will be used to check that the target and features have the same dimensions and indices.
- **y** (*pd.Series, np.ndarray*) – Target data to check for invalid values.

**Returns** List with DataCheckErrors if any invalid values are found in the target data.

**Return type** dict (*DataCheckError*)

### Examples

```
>>> import pandas as pd
```

Target values must be integers, doubles, or booleans.

```
>>> X = pd.DataFrame({"col": [1, 2, 3, 1]})
>>> y = pd.Series(["cat_1", "cat_2", "cat_1", "cat_2"])
>>> target_check = InvalidTargetDataCheck("regression", "R2", null_strategy=
↪"impute")
>>> assert target_check.validate(X, y) == [
...     {
...         "message": "Target is unsupported Unknown type. Valid Woodwork␣
↪logical types include: integer, double, boolean, age, age_fractional, integer_
↪nullable, boolean_nullable, age_nullable",
...         "data_check_name": "InvalidTargetDataCheck",
...         "level": "error",
...         "details": {"columns": None, "rows": None, "unsupported_type":
↪"unknown"},
```

(continues on next page)

(continued from previous page)

```
...         "code": "TARGET_UNSUPPORTED_TYPE",
...         "action_options": []
...     },
...     {
...         "message": "Target data type should be numeric for regression type␣
↪problems.",
...         "data_check_name": "InvalidTargetDataCheck",
...         "level": "error",
...         "details": {"columns": None, "rows": None},
...         "code": "TARGET_UNSUPPORTED_TYPE_REGRESSION",
...         "action_options": []
...     }
... ]
```

The target cannot have null values.

```
>>> y = pd.Series([None, pd.NA, pd.NaT, None])
>>> assert target_check.validate(X, y) == [
...     {
...         "message": "Target is either empty or fully null.",
...         "data_check_name": "InvalidTargetDataCheck",
...         "level": "error",
...         "details": {"columns": None, "rows": None},
...         "code": "TARGET_IS_EMPTY_OR_FULLY_NULL",
...         "action_options": []
...     }
... ]
...
...
>>> y = pd.Series([1, None, 3, None])
>>> assert target_check.validate(None, y) == [
...     {
...         "message": "2 row(s) (50.0%) of target values are null",
...         "data_check_name": "InvalidTargetDataCheck",
...         "level": "error",
...         "details": {
...             "columns": None,
...             "rows": [1, 3],
...             "num_null_rows": 2,
...             "pct_null_rows": 50.0
...         },
...         "code": "TARGET_HAS_NULL",
...         "action_options": [
...            {
...                 "code": "IMPUTE_COL",
...                 "data_check_name": "InvalidTargetDataCheck",
...                 "parameters": {
...                     "impute_strategy": {
...                         "parameter_type": "global",
...                         "type": "category",
...                         "categories": ["mean", "most_frequent"],
...                         "default_value": "mean"
```

(continues on next page)

(continued from previous page)

```
...                     }
...                 },
...                 "metadata": {"columns": None, "rows": None, "is_target":␣
↪True},
...             }
...         ],
...     }
... ]
```

If the target values don't match the problem type passed, an error will be raised. In this instance, only two values exist in the target column, but multiclass has been passed as the problem type.

```
>>> X = pd.DataFrame([i for i in range(50)])
>>> y = pd.Series([i%2 for i in range(50)])
>>> target_check = InvalidTargetDataCheck("multiclass", "Log Loss Multiclass")
>>> assert target_check.validate(X, y) == [
...     {
...         "message": "Target has two or less classes, which is too few for␣
↪multiclass problems.  Consider changing to binary.",
...         "data_check_name": "InvalidTargetDataCheck",
...         "level": "error",
...         "details": {"columns": None, "rows": None, "num_classes": 2},
...         "code": "TARGET_MULTICLASS_NOT_ENOUGH_CLASSES",
...         "action_options": []
...     }
... ]
```

If the length of X and y differ, a warning will be raised. A warning will also be raised for indices that don"t match.

```
>>> target_check = InvalidTargetDataCheck("regression", "R2")
>>> X = pd.DataFrame([i for i in range(5)])
>>> y = pd.Series([1, 2, 4, 3], index=[1, 2, 4, 3])
>>> assert target_check.validate(X, y) == [
...     {
...         "message": "Input target and features have different lengths",
...         "data_check_name": "InvalidTargetDataCheck",
...         "level": "warning",
...         "details": {"columns": None, "rows": None, "features_length": 5,
↪"target_length": 4},
...         "code": "MISMATCHED_LENGTHS",
...         "action_options": []
...     },
...     {
...         "message": "Input target and features have mismatched indices.␣
↪Details will include the first 10 mismatched indices.",
...         "data_check_name": "InvalidTargetDataCheck",
...         "level": "warning",
...         "details": {
...             "columns": None,
...             "rows": None,
...             "indices_not_in_features": [],
```

(continues on next page)

(continued from previous page)

```
...                 "indices_not_in_target": [0]
...             },
...             "code": "MISMATCHED_INDICES",
...             "action_options": []
...         }
... ]
```

**class** evalml.data_checks.**MulticollinearityDataCheck**(*threshold=0.9*)

Check if any set features are likely to be multicollinear.

**Parameters** **threshold** (*float*) – The threshold to be considered. Defaults to 0.9.

**Methods**

| | |
|---|---|
| *name* | Return a name describing the data check. |
| *validate* | Check if any set of features are likely to be multicollinear. |

**name**(*cls*)

Return a name describing the data check.

**validate**(*self*, *X*, *y=None*)

Check if any set of features are likely to be multicollinear.

**Parameters**

- **X** (*pd.DataFrame*) – The input features to check.
- **y** (*pd.Series*) – The target. Ignored.

**Returns** dict with a DataCheckWarning if there are any potentially multicollinear columns.

**Return type** dict

### Example

```
>>> import pandas as pd
```

Columns in X that are highly correlated with each other will be identified using mutual information.

```
>>> col = pd.Series([1, 0, 2, 3, 4] * 15)
>>> X = pd.DataFrame({"col_1": col, "col_2": col * 3})
>>> y = pd.Series([1, 0, 0, 1, 0] * 15)
...
>>> multicollinearity_check = MulticollinearityDataCheck(threshold=1.0)
>>> assert multicollinearity_check.validate(X, y) == [
...     {
...         "message": "Columns are likely to be correlated: [('col_1', 'col_2
↪')]",
...         "data_check_name": "MulticollinearityDataCheck",
...         "level": "warning",
...         "code": "IS_MULTICOLLINEAR",
...         "details": {"columns": [("col_1", "col_2")], "rows": None},
...         "action_options": []
```

(continues on next page)

(continued from previous page)

```
...     }
... ]
```

**class** evalml.data_checks.**NoVarianceDataCheck**(*count_nan_as_value=False*)

Check if the target or any of the features have no variance.

**Parameters** **count_nan_as_value** (*bool*) – If True, missing values will be counted as their own unique value. Additionally, if true, will return a DataCheckWarning instead of an error if the feature has mostly missing data and only one unique value. Defaults to False.

**Methods**

| | |
|---|---|
| *name* | Return a name describing the data check. |
| *validate* | Check if the target or any of the features have no variance (1 unique value). |

**name**(*cls*)

Return a name describing the data check.

**validate**(*self, X, y=None*)

Check if the target or any of the features have no variance (1 unique value).

**Parameters**

- **X** (*pd.DataFrame, np.ndarray*) – The input features.
- **y** (*pd.Series, np.ndarray*) – Optional, the target data.

**Returns** A dict of warnings/errors corresponding to features or target with no variance.

**Return type** dict

### Examples

```
>>> import pandas as pd
```

Columns or target data that have only one unique value will raise an error.

```
>>> X = pd.DataFrame([2, 2, 2, 2, 2, 2, 2, 2], columns=["First_Column"])
>>> y = pd.Series([1, 1, 1, 1, 1, 1, 1, 1])
...
>>> novar_dc = NoVarianceDataCheck()
>>> assert novar_dc.validate(X, y) == [
...     {
...         "message": "'First_Column' has 1 unique value.",
...         "data_check_name": "NoVarianceDataCheck",
...         "level": "warning",
...         "details": {"columns": ["First_Column"], "rows": None},
...         "code": "NO_VARIANCE",
...         "action_options": [
...             {
...                 "code": "DROP_COL",
...                 "data_check_name": "NoVarianceDataCheck",
...                 "parameters": {},
```

(continues on next page)

(continued from previous page)

```
...                 "metadata": {"columns": ["First_Column"], "rows": None}
...             },
...         ]
...     },
...     {
...         "message": "Y has 1 unique value.",
...         "data_check_name": "NoVarianceDataCheck",
...         "level": "warning",
...         "details": {"columns": ["Y"], "rows": None},
...         "code": "NO_VARIANCE",
...         "action_options": []
...     }
... ]
```

By default, NaNs will not be counted as distinct values. In the first example, there are still two distinct values besides None. In the second, there are no distinct values as the target is entirely null.

```
>>> X["First_Column"] = [2, 2, 2, 3, 3, 3, None, None]
>>> y = pd.Series([1, 1, 1, 2, 2, 2, None, None])
>>> assert novar_dc.validate(X, y) == []
...
...
>>> y = pd.Series([None] * 7)
>>> assert novar_dc.validate(X, y) == [
...     {
...         "message": "Y has 0 unique values.",
...         "data_check_name": "NoVarianceDataCheck",
...         "level": "warning",
...         "details": {"columns": ["Y"], "rows": None},
...         "code": "NO_VARIANCE_ZERO_UNIQUE",
...         "action_options":[]
...     }
... ]
```

As None is not considered a distinct value by default, there is only one unique value in X and y.

```
>>> X["First_Column"] = [2, 2, 2, 2, None, None, None, None]
>>> y = pd.Series([1, 1, 1, 1, None, None, None, None])
>>> assert novar_dc.validate(X, y) == [
...     {
...         "message": "'First_Column' has 1 unique value.",
...         "data_check_name": "NoVarianceDataCheck",
...         "level": "warning",
...         "details": {"columns": ["First_Column"], "rows": None},
...         "code": "NO_VARIANCE",
...         "action_options": [
...             {
...                 "code": "DROP_COL",
...                  "data_check_name": "NoVarianceDataCheck",
...                  "parameters": {},
...                  "metadata": {"columns": ["First_Column"], "rows": None}
...             },
```

(continues on next page)

(continued from previous page)

```
...         ]
...     },
...     {
...         "message": "Y has 1 unique value.",
...         "data_check_name": "NoVarianceDataCheck",
...         "level": "warning",
...         "details": {"columns": ["Y"], "rows": None},
...         "code": "NO_VARIANCE",
...         "action_options": []
...     }
... ]
```

If count_nan_as_value is set to True, then NaNs are counted as unique values. In the event that there is an adequate number of unique values only because count_nan_as_value is set to True, a warning will be raised so the user can encode these values.

```
>>> novar_dc = NoVarianceDataCheck(count_nan_as_value=True)
>>> assert novar_dc.validate(X, y) == [
...     {
...         "message": "'First_Column' has two unique values including nulls.␣
↪Consider encoding the nulls for this column to be useful for machine learning.
↪",
...         "data_check_name": "NoVarianceDataCheck",
...         "level": "warning",
...         "details": {"columns": ["First_Column"], "rows": None},
...         "code": "NO_VARIANCE_WITH_NULL",
...         "action_options": [
...             {
...                 "code": "DROP_COL",
...                  "data_check_name": "NoVarianceDataCheck",
...                  "parameters": {},
...                  "metadata": {"columns": ["First_Column"], "rows": None}
...             },
...         ]
...     },
...     {
...         "message": "Y has two unique values including nulls. Consider␣
↪encoding the nulls for this column to be useful for machine learning.",
...         "data_check_name": "NoVarianceDataCheck",
...         "level": "warning",
...         "details": {"columns": ["Y"], "rows": None},
...         "code": "NO_VARIANCE_WITH_NULL",
...         "action_options": []
...     }
... ]
```

**class** evalml.data_checks.**NullDataCheck**(*pct_null_col_threshold=0.95*, *pct_moderately_null_col_threshold=0.2*, *pct_null_row_threshold=0.95*)

Check if there are any highly-null numerical, boolean, categorical, natural language, and unknown columns and rows in the input.

**Parameters**

- **pct_null_col_threshold** (*float*) – If the percentage of NaN values in an input feature exceeds this amount, that column will be considered highly-null. Defaults to 0.95.
- **pct_moderately_null_col_threshold** (*float*) – If the percentage of NaN values in an input feature exceeds this amount but is less than the percentage specified in pct_null_col_threshold, that column will be considered moderately-null. Defaults to 0.20.
- **pct_null_row_threshold** (*float*) – If the percentage of NaN values in an input row exceeds this amount, that row will be considered highly-null. Defaults to 0.95.

**Methods**

| | |
|---|---|
| *get_null_column_information* | Finds columns that are considered highly null (percentage null is greater than threshold) and returns dictionary mapping column name to percentage null and dictionary mapping column name to null indices. |
| *get_null_row_information* | Finds rows that are considered highly null (percentage null is greater than threshold). |
| *name* | Return a name describing the data check. |
| *validate* | Check if there are any highly-null columns or rows in the input. |

**static get_null_column_information**(*X*, *pct_null_col_threshold=0.0*)

Finds columns that are considered highly null (percentage null is greater than threshold) and returns dictionary mapping column name to percentage null and dictionary mapping column name to null indices.

**Parameters**

- **X** (*pd.DataFrame*) – DataFrame to check for highly null columns.
- **pct_null_col_threshold** (*float*) – Percentage threshold for a column to be considered null. Defaults to 0.0.

**Returns** Tuple containing: dictionary mapping column name to its null percentage and dictionary mapping column name to null indices in that column.

**Return type** tuple

**static get_null_row_information**(*X*, *pct_null_row_threshold=0.0*)

Finds rows that are considered highly null (percentage null is greater than threshold).

**Parameters**

- **X** (*pd.DataFrame*) – DataFrame to check for highly null rows.
- **pct_null_row_threshold** (*float*) – Percentage threshold for a row to be considered null. Defaults to 0.0.

**Returns** Series containing the percentage null for each row.

**Return type** pd.Series

**name**(*cls*)

Return a name describing the data check.

**validate**(*self*, *X*, *y=None*)

Check if there are any highly-null columns or rows in the input.

**Parameters**

- **X** (*pd.DataFrame, np.ndarray*) – Features.

- **y** (*pd.Series, np.ndarray*) – Ignored. Defaults to None.

**Returns** dict with a DataCheckWarning if there are any highly-null columns or rows.

**Return type** dict

### Examples

```
>>> import pandas as pd
...
>>> class SeriesWrap():
...     def __init__(self, series):
...         self.series = series
...
...     def __eq__(self, series_2):
...         return all(self.series.eq(series_2.series))
```

With pct_null_col_threshold set to 0.50, any column that has 50% or more of its observations set to null will be included in the warning, as well as the percentage of null values identified ("all_null": 1.0, "lots_of_null": 0.8).

```
>>> df = pd.DataFrame({
...     "all_null": [None, pd.NA, None, None, None],
...     "lots_of_null": [None, None, None, None, 5],
...     "few_null": [1, 2, None, 2, 3],
...     "no_null": [1, 2, 3, 4, 5]
... })
...
>>> highly_null_dc = NullDataCheck(pct_null_col_threshold=0.50)
>>> assert highly_null_dc.validate(df) == [
...     {
...         "message": "Column(s) 'all_null', 'lots_of_null' are 50.0% or more␣
↪null",
...         "data_check_name": "NullDataCheck",
...         "level": "warning",
...         "details": {
...             "columns": ["all_null", "lots_of_null"],
...             "rows": None,
...             "pct_null_rows": {"all_null": 1.0, "lots_of_null": 0.8}
...         },
...         "code": "HIGHLY_NULL_COLS",
...         "action_options": [
...             {
...                 "code": "DROP_COL",
...                 "data_check_name": "NullDataCheck",
...                 "parameters": {},
...                 "metadata": {"columns": ["all_null", "lots_of_null"], "rows
↪": None}
...             }
...         ]
...     },
...     {
...         "message": "Column(s) 'few_null' have between 20.0% and 50.0% null␣
↪values",
```

(continues on next page)

(continued from previous page)

```
...         "data_check_name": "NullDataCheck",
...         "level": "warning",
...         "details": {"columns": ["few_null"], "rows": None},
...         "code": "COLS_WITH_NULL",
...         "action_options": [
...             {
...                 "code": "IMPUTE_COL",
...                 "data_check_name": "NullDataCheck",
...                 "metadata": {"columns": ["few_null"], "rows": None, "is_
↪target": False},
...                 "parameters": {
...                     "impute_strategies": {
...                         "parameter_type": "column",
...                         "columns": {
...                             "few_null": {
...                                 "impute_strategy": {"categories": ["mean",
↪"most_frequent"], "type": "category", "default_value": "mean"}
...                             }
...                         }
...                     }
...                 }
...             }
...         ]
...     }
... ]
```

With pct_null_row_threshold set to 0.50, any row with 50% or more of its respective column values set to null will included in the warning, as well as the offending rows ("rows": [0, 1, 2, 3]). Since the default value for pct_null_col_threshold is 0.95, "all_null" is also included in the warnings since the percentage of null values in that row is over 95%. Since the default value for pct_moderately_null_col_threshold is 0.20, "few_null" is included as a "moderately null" column as it has a null column percentage of 20%.

```
>>> highly_null_dc = NullDataCheck(pct_null_row_threshold=0.50)
>>> validation_messages = highly_null_dc.validate(df)
>>> validation_messages[0]["details"]["pct_null_cols"] = SeriesWrap(validation_
↪messages[0]["details"]["pct_null_cols"])
>>> highly_null_rows = SeriesWrap(pd.Series([0.5, 0.5, 0.75, 0.5]))
>>> assert validation_messages == [
...     {
...         "message": "4 out of 5 rows are 50.0% or more null",
...         "data_check_name": "NullDataCheck",
...         "level": "warning",
...         "details": {
...             "columns": None,
...             "rows": [0, 1, 2, 3],
...             "pct_null_cols": highly_null_rows
...         },
...         "code": "HIGHLY_NULL_ROWS",
...         "action_options": [
...             {
...                 "code": "DROP_ROWS",
...                  "data_check_name": "NullDataCheck",
```

(continues on next page)

(continued from previous page)

```
...                     "parameters": {},
...                     "metadata": {"columns": None, "rows": [0, 1, 2, 3]}
...                 }
...             ]
...         },
...         {
...             "message": "Column(s) 'all_null' are 95.0% or more null",
...             "data_check_name": "NullDataCheck",
...             "level": "warning",
...             "details": {
...                 "columns": ["all_null"],
...                 "rows": None,
...                 "pct_null_rows": {"all_null": 1.0}
...             },
...            "code": "HIGHLY_NULL_COLS",
...            "action_options": [
...                 {
...                     "code": "DROP_COL",
...                     "data_check_name": "NullDataCheck",
...                     "metadata": {"columns": ["all_null"], "rows": None},
...                     "parameters": {}
...                 }
...             ]
...         },
...         {
...             "message": "Column(s) 'lots_of_null', 'few_null' have between 20.0%␣
↪and 95.0% null values",
...             "data_check_name": "NullDataCheck",
...             "level": "warning",
...             "details": {"columns": ["lots_of_null", "few_null"], "rows": None},
...             "code": "COLS_WITH_NULL",
...             "action_options": [
...                 {
...                    "code": "IMPUTE_COL",
...                    "data_check_name": "NullDataCheck",
...                    "metadata": {"columns": ["lots_of_null", "few_null"], "rows":
↪ None, "is_target": False},
...                    "parameters": {
...                        "impute_strategies": {
...                            "parameter_type": "column",
...                            "columns": {
...                                "lots_of_null": {"impute_strategy": {"categories
↪": ["mean", "most_frequent"], "type": "category", "default_value": "mean"}},
...                                "few_null": {"impute_strategy": {"categories": [
↪"mean", "most_frequent"], "type": "category", "default_value": "mean"}}
...                            }
...                        }
...                    }
...                 }
...             ]
...         }
... ]
```

**class** evalml.data_checks.**OutliersDataCheck**

Checks if there are any outliers in input data by using IQR to determine score anomalies.

Columns with score anomalies are considered to contain outliers.

**Methods**

| | |
|---|---|
| *get_boxplot_data* | Returns box plot information for the given data. |
| *name* | Return a name describing the data check. |
| *validate* | Check if there are any outliers in a dataframe by using IQR to determine column anomalies. Column with anomalies are considered to contain outliers. |

**static get_boxplot_data**(*data_*)

Returns box plot information for the given data.

**Parameters** **data** (*pd.Series, np.ndarray*) – Input data.

**Returns** A payload of box plot statistics.

**Return type** dict

### Examples

```
>>> import pandas as pd
...
>>> df = pd.DataFrame({
...     "x": [1, 2, 3, 4, 5],
...     "y": [6, 7, 8, 9, 10],
...     "z": [-1, -2, -3, -1201, -4]
... })
>>> box_plot_data = OutliersDataCheck.get_boxplot_data(df["z"])
>>> box_plot_data["score"] = round(box_plot_data["score"], 2)
>>> assert box_plot_data == {
...     "score": 0.89,
...     "pct_outliers": 0.2,
...     "values": {"q1": -4.0,
...                "median": -3.0,
...                "q3": -2.0,
...                "low_bound": -7.0,
...                "high_bound": -1.0,
...                "low_values": [-1201],
...                "high_values": [],
...                "low_indices": [3],
...                "high_indices": []}
...     }
```

**name**(*cls*)

Return a name describing the data check.

**validate**(*self, X, y=None*)

Check if there are any outliers in a dataframe by using IQR to determine column anomalies. Column with anomalies are considered to contain outliers.

**Parameters**

- **X** (*pd.DataFrame, np.ndarray*) – Input features.
- **y** (*pd.Series, np.ndarray*) – Ignored. Defaults to None.

**Returns** A dictionary with warnings if any columns have outliers.

**Return type** dict

### Examples

```
>>> import pandas as pd
```

The column "z" has an outlier so a warning is added to alert the user of its location.

```
>>> df = pd.DataFrame({
...     "x": [1, 2, 3, 4, 5],
...     "y": [6, 7, 8, 9, 10],
...     "z": [-1, -2, -3, -1201, -4]
... })
...
>>> outliers_check = OutliersDataCheck()
>>> assert outliers_check.validate(df) == [
...     {
...         "message": "Column(s) 'z' are likely to have outlier data.",
...         "data_check_name": "OutliersDataCheck",
...         "level": "warning",
...         "code": "HAS_OUTLIERS",
...         "details": {"columns": ["z"], "rows": [3], "column_indices": {"z":␣
↪[3]}},
...         "action_options": [
...             {
...                 "code": "DROP_ROWS",
...                  "data_check_name": "OutliersDataCheck",
...                  "parameters": {},
...                  "metadata": {"rows": [3], "columns": None}
...             }
...         ]
...     }
... ]
```

**class** evalml.data_checks.**SparsityDataCheck**(*problem_type*, *threshold*, *unique_count_threshold=10*)

Check if there are any columns with sparsely populated values in the input.

**Parameters**

- **problem_type** (*str or ProblemTypes*) – The specific problem type to data check for. 'multiclass' or 'time series multiclass' is the only accepted problem type.
- **threshold** (*float*) – The threshold value, or percentage of each column's unique values, below which, a column exhibits sparsity. Should be between 0 and 1.
- **unique_count_threshold** (*int*) – The minimum number of times a unique value has to be present in a column to not be considered "sparse." Defaults to 10.

**Methods**

| | |
|---|---|
| *name* | Return a name describing the data check. |
| *sparsity_score* | Calculate a sparsity score for the given value counts by calculating the percentage of unique values that exceed the count_threshold. |
| *validate* | Calculate what percentage of each column's unique values exceed the count threshold and compare that percentage to the sparsity threshold stored in the class instance. |

**name**(*cls*)

Return a name describing the data check.

**static sparsity_score**(*col*, *count_threshold=10*)

Calculate a sparsity score for the given value counts by calculating the percentage of unique values that exceed the count_threshold.

**Parameters**

- **col** (*pd.Series*) – Feature values.
- **count_threshold** (*int*) – The number of instances below which a value is considered sparse. Default is 10.

**Returns** Sparsity score, or the percentage of the unique values that exceed count_threshold.

**Return type** (float)

**validate**(*self*, *X*, *y=None*)

Calculate what percentage of each column's unique values exceed the count threshold and compare that percentage to the sparsity threshold stored in the class instance.

**Parameters**

- **X** (*pd.DataFrame, np.ndarray*) – Features.
- **y** (*pd.Series, np.ndarray*) – Ignored.

**Returns** dict with a DataCheckWarning if there are any sparse columns.

**Return type** dict

### Examples

```
>>> import pandas as pd
```

For multiclass problems, if a column doesn't have enough representation from unique values, it will be considered sparse.

```
>>> df = pd.DataFrame({
...     "sparse": [float(x) for x in range(100)],
...     "not_sparse": [float(1) for x in range(100)]
... })
...
>>> sparsity_check = SparsityDataCheck(problem_type="multiclass", threshold=0.5,
↪ unique_count_threshold=10)
>>> assert sparsity_check.validate(df) == [
...     {
```

(continues on next page)

(continued from previous page)

```
...         "message": "Input columns ('sparse') for multiclass problem type␣
↪are too sparse.",
...         "data_check_name": "SparsityDataCheck",
...         "level": "warning",
...         "code": "TOO_SPARSE",
...         "details": {
...             "columns": ["sparse"],
...             "sparsity_score": {"sparse": 0.0},
...             "rows": None
...         },
...         "action_options": [
...             {
...                 "code": "DROP_COL",
...                  "data_check_name": "SparsityDataCheck",
...                  "parameters": {},
...                  "metadata": {"columns": ["sparse"], "rows": None}
...             }
...         ]
...     }
... ]
```

... >>> df["sparse"] = [float(x % 10) for x in range(100)] >>> sparsity_check = SparsityDataCheck(problem_type="multiclass", threshold=1, unique_count_threshold=5) >>> assert sparsity_check.validate(df) == [] ... >>> sparse_array = pd.Series([1, 1, 1, 2, 2, 3] * 3) >>> assert SparsityDataCheck.sparsity_score(sparse_array, count_threshold=5) == 0.6666666666666666

**class** `evalml.data_checks.`**`TargetDistributionDataCheck`**

Check if the target data contains certain distributions that may need to be transformed prior training to improve model performance. Uses the Shapiro-Wilks test when the dataset is <=5000 samples, otherwise uses Jarque-Bera.

**Methods**

| | |
|---|---|
| *name* | Return a name describing the data check. |
| *validate* | Check if the target data has a certain distribution. |

**name**(*cls*)

Return a name describing the data check.

**validate**(*self*, *X*, *y*)

Check if the target data has a certain distribution.

**Parameters**

- **X** (*pd.DataFrame, np.ndarray*) – Features. Ignored.
- **y** (*pd.Series, np.ndarray*) – Target data to check for underlying distributions.

**Returns** List with DataCheckErrors if certain distributions are found in the target data.

**Return type** dict (*DataCheckError*)

### Examples

```
>>> import pandas as pd
```

Targets that exhibit a lognormal distribution will raise a warning for the user to transform the target.

```
>>> y = [0.946, 0.972, 1.154, 0.954, 0.969, 1.222, 1.038, 0.999, 0.973, 0.897]
>>> target_check = TargetDistributionDataCheck()
>>> assert target_check.validate(None, y) == [
...     {
...         "message": "Target may have a lognormal distribution.",
...         "data_check_name": "TargetDistributionDataCheck",
...         "level": "warning",
...         "code": "TARGET_LOGNORMAL_DISTRIBUTION",
...         "details": {"normalization_method": "shapiro", "statistic": 0.8, "p-
↪value": 0.045, "columns": None, "rows": None},
...         "action_options": [
...             {
...                 "code": "TRANSFORM_TARGET",
...                 "data_check_name": "TargetDistributionDataCheck",
...                 "parameters": {},
...                 "metadata": {
...                     "transformation_strategy": "lognormal",
...                     "is_target": True,
...                     "columns": None,
...                     "rows": None
...                 }
...             }
...         ]
...     }
... ]
...
>>> y = pd.Series([1, 1, 1, 2, 2, 3, 4, 4, 5, 5, 5])
>>> assert target_check.validate(None, y) == []
...
...
>>> y = pd.Series(pd.date_range("1/1/21", periods=10))
>>> assert target_check.validate(None, y) == [
...     {
...         "message": "Target is unsupported datetime type. Valid Woodwork␣
↪logical types include: integer, double, age, age_fractional",
...         "data_check_name": "TargetDistributionDataCheck",
...         "level": "error",
...         "details": {"columns": None, "rows": None, "unsupported_type":
↪"datetime"},
...         "code": "TARGET_UNSUPPORTED_TYPE",
...         "action_options": []
...     }
... ]
```

**class** evalml.data_checks.**TargetLeakageDataCheck**(*pct_corr_threshold=0.95*, *method='all'*)

Check if any of the features are highly correlated with the target by using mutual information, Pearson correlation, and other correlation metrics.

If method='mutual_info', this data check uses mutual information and supports all target and feature types. Other correlation metrics only support binary with numeric and boolean dtypes. This method will return a value in [-1, 1] if other correlation metrics are selected and will returns a value in [0, 1] if mutual information is selected. Correlation metrics available can be found in Woodwork's dependence_dict method.

**Parameters**

- **pct_corr_threshold** (*float*) – The correlation threshold to be considered leakage. Defaults to 0.95.
- **method** (*string*) – The method to determine correlation. Use 'all' or 'max' for the maximum correlation, or for specific correlation metrics, use their name (ie 'mutual_info' for mutual information, 'pearson' for Pearson correlation, etc). possible methods can be found in Woodwork's config, under *correlation_metrics*. Defaults to 'all'.

**Methods**

| | |
|---|---|
| *name* | Return a name describing the data check. |
| *validate* | Check if any of the features are highly correlated with the target by using mutual information, Pearson correlation, and/or Spearman correlation. |

**name**(*cls*)

Return a name describing the data check.

**validate**(*self*, *X*, *y*)

Check if any of the features are highly correlated with the target by using mutual information, Pearson correlation, and/or Spearman correlation.

If *method='mutual_info'* or *'method='max'*, supports all target and feature types. Other correlation metrics only support binary with numeric and boolean dtypes. This method will return a value in [-1, 1] if other correlation metrics are selected and will returns a value in [0, 1] if mutual information is selected.

**Parameters**

- **X** (*pd.DataFrame, np.ndarray*) – The input features to check.
- **y** (*pd.Series, np.ndarray*) – The target data.

**Returns** dict with a DataCheckWarning if target leakage is detected.

**Return type** dict (*DataCheckWarning*)

### Examples

```
>>> import pandas as pd
```

Any columns that are strongly correlated with the target will raise a warning. This could be indicative of data leakage.

```
>>> X = pd.DataFrame({
...     "leak": [10, 42, 31, 51, 61] * 15,
...     "x": [42, 54, 12, 64, 12] * 15,
...     "y": [13, 5, 13, 74, 24] * 15,
... })
>>> y = pd.Series([10, 42, 31, 51, 40] * 15)
...
```

(continues on next page)

(continued from previous page)

```
>>> target_leakage_check = TargetLeakageDataCheck(pct_corr_threshold=0.95)
>>> assert target_leakage_check.validate(X, y) == [
...     {
...         "message": "Column 'leak' is 95.0% or more correlated with the␣
↪target",
...         "data_check_name": "TargetLeakageDataCheck",
...         "level": "warning",
...         "code": "TARGET_LEAKAGE",
...         "details": {"columns": ["leak"], "rows": None},
...         "action_options": [
...             {
...                 "code": "DROP_COL",
...                 "data_check_name": "TargetLeakageDataCheck",
...                 "parameters": {},
...                 "metadata": {"columns": ["leak"], "rows": None}
...             }
...         ]
...     }
... ]
```

The default method can be changed to pearson from mutual_info.

```
>>> X["x"] = y / 2
>>> target_leakage_check = TargetLeakageDataCheck(pct_corr_threshold=0.8,␣
↪method="pearson")
>>> assert target_leakage_check.validate(X, y) == [
...     {
...         "message": "Columns 'leak', 'x' are 80.0% or more correlated with␣
↪the target",
...         "data_check_name": "TargetLeakageDataCheck",
...         "level": "warning",
...         "details": {"columns": ["leak", "x"], "rows": None},
...         "code": "TARGET_LEAKAGE",
...         "action_options": [
...             {
...                 "code": "DROP_COL",
...                  "data_check_name": "TargetLeakageDataCheck",
...                  "parameters": {},
...                  "metadata": {"columns": ["leak", "x"], "rows": None}
...             }
...         ]
...     }
... ]
```

**class** evalml.data_checks.**TimeSeriesParametersDataCheck**(*problem_configuration*, *n_splits*)

Checks whether the time series parameters are compatible with data splitting.

If *gap* + *max_delay* + *forecast_horizon* > *X.shape[0] // (n_splits + 1)*

then the feature engineering window is larger than the smallest split. This will cause the pipeline to create features from data that does not exist, which will cause errors.

**Parameters**

- **problem_configuration** (*dict*) – Dict containing problem_configuration parameters.

- **n_splits** (*int*) – Number of time series splits.

**Methods**

| | |
|---|---|
| *name* | Return a name describing the data check. |
| *validate* | Check if the time series parameters are compatible with data splitting. |

**name**(*cls*)

Return a name describing the data check.

**validate**(*self*, *X*, *y=None*)

Check if the time series parameters are compatible with data splitting.

**Parameters**

- **X** (*pd.DataFrame, np.ndarray*) – Features.
- **y** (*pd.Series, np.ndarray*) – Ignored. Defaults to None.

**Returns** dict with a DataCheckError if parameters are too big for the split sizes.

**Return type** dict

### Examples

```
>>> import pandas as pd
```

The time series parameters have to be compatible with the data passed. If the window size (gap + max_delay + forecast_horizon) is greater than or equal to the split size, then an error will be raised.

```
>>> X = pd.DataFrame({
...     "dates": pd.date_range("1/1/21", periods=100),
...     "first": [i for i in range(100)],
... })
>>> y = pd.Series([i for i in range(100)])
...
>>> problem_config = {"gap": 7, "max_delay": 2, "forecast_horizon": 12, "time_
↪index": "dates"}
>>> ts_parameters_check = TimeSeriesParametersDataCheck(problem_
↪configuration=problem_config, n_splits=7)
>>> assert ts_parameters_check.validate(X, y) == [
...     {
...         "message": "Since the data has 100 observations, n_splits=7, and a␣
↪forecast horizon of 12, the smallest "
...                    "split would have 16 observations. Since 21 (gap + max_
↪delay + forecast_horizon)"
...                    " >= 16, then at least one of the splits would be empty␣
↪by the time it reaches "
...                    "the pipeline. Please use a smaller number of splits,␣
↪reduce one or more these "
...                    "parameters, or collect more data.",
...         "data_check_name": "TimeSeriesParametersDataCheck",
...         "level": "error",
...         "code": "TIMESERIES_PARAMETERS_NOT_COMPATIBLE_WITH_SPLIT",
```

(continues on next page)

(continued from previous page)

```
...         "details": {
...             "columns": None,
...             "rows": None,
...             "max_window_size": 21,
...             "min_split_size": 16,
...             "n_obs": 100,
...             "n_splits": 7
...         },
...         "action_options": []
...     }
... ]
```

**class** evalml.data_checks.**TimeSeriesSplittingDataCheck**(*problem_type*, *n_splits*)

Checks whether the time series target data is compatible with splitting.

If the target data in the training and validation of every split doesn't have representation from all classes (for time series classification problems) this will prevent the estimators from training on all potential outcomes which will cause errors during prediction.

**Parameters**

- **problem_type** (*str or ProblemTypes*) – Problem type.
- **n_splits** (*int*) – Number of time series splits.

**Methods**

| | |
|---|---|
| *name* | Return a name describing the data check. |
| *validate* | Check if the training and validation targets are compatible with time series data splitting. |

**name**(*cls*)

Return a name describing the data check.

**validate**(*self*, *X*, *y*)

Check if the training and validation targets are compatible with time series data splitting.

**Parameters**

- **X** (*pd.DataFrame, np.ndarray*) – Ignored. Features.
- **y** (*pd.Series, np.ndarray*) – Target data.

**Returns** dict with a DataCheckError if splitting would result in inadequate class representation.

**Return type** dict

### Example

```
>>> import pandas as pd
```

Passing n_splits as 3 means that the data will be segmented into 4 parts to be iterated over for training and validation splits. The first split results in training indices of [0:25] and validation indices of [25:50]. The training indices of the first split result in only one unique value (0). The third split results in training indices of [0:75] and validation indices of [75:100]. The validation indices of the third split result in only one unique value (1).

```
>>> X = None
>>> y = pd.Series([0 if i < 45 else i % 2 if i < 55 else 1 for i in range(100)])
>>> ts_splitting_check = TimeSeriesSplittingDataCheck("time series binary", 3)
>>> assert ts_splitting_check.validate(X, y) == [
...     {
...         "message": "Time Series Binary and Time Series Multiclass problem "
...                    "types require every training and validation split to "
...                    "have at least one instance of all the target classes. "
...                    "The following splits are invalid: [1, 3]",
...         "data_check_name": "TimeSeriesSplittingDataCheck",
...         "level": "error",
...         "details": {
...             "columns": None, "rows": None,
...             "invalid_splits": {
...                 1: {"Training": [0, 25]},
...                 3: {"Validation": [75, 100]}
...             }
...         },
...         "code": "TIMESERIES_TARGET_NOT_COMPATIBLE_WITH_SPLIT",
...         "action_options": []
...     }
... ]
```

**class** evalml.data_checks.**UniquenessDataCheck**(*problem_type*, *threshold=0.5*)

Check if there are any columns in the input that are either too unique for classification problems or not unique enough for regression problems.

**Parameters**

- **problem_type** (*str or ProblemTypes*) – The specific problem type to data check for. e.g. 'binary', 'multiclass', 'regression, 'time series regression'
- **threshold** (*float*) – The threshold to set as an upper bound on uniqueness for classification type problems or lower bound on for regression type problems. Defaults to 0.50.

**Methods**

| | |
|---|---|
| *name* | Return a name describing the data check. |
| *uniqueness_score* | Calculate a uniqueness score for the provided field. NaN values are not considered as unique values in the calculation. |
| *validate* | Check if there are any columns in the input that are too unique in the case of classification problems or not unique enough in the case of regression problems. |

**name**(*cls*)

Return a name describing the data check.

**static uniqueness_score**(*col*, *drop_na=True*)

Calculate a uniqueness score for the provided field. NaN values are not considered as unique values in the calculation.

Based on the Herfindahl-Hirschman Index.

**Parameters**

- **col** (*pd.Series*) – Feature values.
- **drop_na** (*bool*) – Whether to drop null values when computing the uniqueness score. Defaults to True.

**Returns** Uniqueness score.

**Return type** (float)

**validate**(*self*, *X*, *y=None*)

Check if there are any columns in the input that are too unique in the case of classification problems or not unique enough in the case of regression problems.

**Parameters**

- **X** (*pd.DataFrame, np.ndarray*) – Features.
- **y** (*pd.Series, np.ndarray*) – Ignored. Defaults to None.

**Returns**

**dict with a DataCheckWarning if there are any too unique or not** unique enough columns.

**Return type** dict

### Examples

```
>>> import pandas as pd
```

Because the problem type is regression, the column "regression_not_unique_enough" raises a warning for having just one value.

```
>>> df = pd.DataFrame({
...     "regression_unique_enough": [float(x) for x in range(100)],
...     "regression_not_unique_enough": [float(1) for x in range(100)]
... })
...
>>> uniqueness_check = UniquenessDataCheck(problem_type="regression",␣
↪threshold=0.8)
>>> assert uniqueness_check.validate(df) == [
...     {
...         "message": "Input columns 'regression_not_unique_enough' for␣
↪regression problem type are not unique enough.",
...         "data_check_name": "UniquenessDataCheck",
...         "level": "warning",
...         "code": "NOT_UNIQUE_ENOUGH",
...         "details": {"columns": ["regression_not_unique_enough"],
↪"uniqueness_score": {"regression_not_unique_enough": 0.0}, "rows": None}
```

(continues on next page)

(continued from previous page)

```
...         "action_options": [
...             {
...                 "code": "DROP_COL",
...                 "parameters": {},
...                 "data_check_name": "UniquenessDataCheck",
...                 "metadata": {"columns": ["regression_not_unique_enough"],
↪"rows": None}
...             }
...         ]
...     }
... ]
```

For multiclass, the column "regression_unique_enough" has too many unique values and will raise an appropriate warning. >>> y = pd.Series([1, 1, 1, 2, 2, 3, 3, 3]) >>> uniqueness_check = UniquenessDataCheck(problem_type="multiclass", threshold=0.8) >>> assert uniqueness_check.validate(df) == [ … { … "message": "Input columns 'regression_unique_enough' for multiclass problem type are too unique.", … "data_check_name": "UniquenessDataCheck", … "level": "warning", … "details": { … "columns": ["regression_unique_enough"], … "rows": None, … "uniqueness_score": {"regression_unique_enough": 0.99} … }, … "code": "TOO_UNIQUE", … "action_options": [ … { … "code": "DROP_COL", … "data_check_name": "UniquenessDataCheck", … "parameters": {}, … "metadata": {"columns": ["regression_unique_enough"], "rows": None} … } … ] … } … ] … >>> assert UniquenessDataCheck.uniqueness_score(y) == 0.65625

### Demos

Demo datasets.

### Submodules

### breast_cancer

Load the breast cancer dataset, which can be used for binary classification problems.

### Module Contents

### Functions

| | |
|---|---|
| *load_breast_cancer* | Load breast cancer dataset. Binary classification problem. |

### Contents

`evalml.demos.breast_cancer.load_breast_cancer()`

Load breast cancer dataset. Binary classification problem.

**Returns** X and y

**Return type** (pd.Dataframe, pd.Series)

## churn

Load the churn dataset, which can be used for binary classification problems.

### Module Contents

### Functions

| | |
|---|---|
| *load_churn* | Load churn dataset, which can be used for binary classification problems. |

### Contents

`evalml.demos.churn.load_churn(n_rows=None, verbose=True)`

Load churn dataset, which can be used for binary classification problems.

**Parameters**

- **n_rows** (*int*) – Number of rows from the dataset to return
- **verbose** (*bool*) – Whether to print information about features and labels

**Returns** X and y

**Return type** (pd.Dataframe, pd.Series)

## diabetes

Load the diabetes dataset, which can be used for regression problems.

### Module Contents

### Functions

| | |
|---|---|
| *load_diabetes* | Load diabetes dataset. Used for regression problem. |

### Contents

`evalml.demos.diabetes.load_diabetes()`

Load diabetes dataset. Used for regression problem.

**Returns** X and y

**Return type** (pd.Dataframe, pd.Series)

### fraud

Load the credit card fraud dataset, which can be used for binary classification problems.

### Module Contents

### Functions

| | |
|---|---|
| *load_fraud* | Load credit card fraud dataset. |

### Contents

`evalml.demos.fraud.load_fraud(n_rows=None, verbose=True)`

Load credit card fraud dataset.

The fraud dataset can be used for binary classification problems.

**Parameters**

- **n_rows** (*int*) – Number of rows from the dataset to return
- **verbose** (*bool*) – Whether to print information about features and labels

**Returns** X and y

**Return type** (pd.Dataframe, pd.Series)

### weather

The Australian daily-min-termperatures weather dataset.

### Module Contents

### Functions

| | |
|---|---|
| *load_weather* | Load the Australian daily-min-termperatures weather dataset. |

### Contents

`evalml.demos.weather.load_weather()`

Load the Australian daily-min-termperatures weather dataset.

**Returns** X and y

**Return type** (pd.Dataframe, pd.Series)

## wine

Load and return the wine dataset, which can be used for multiclass classification problems.

### Module Contents

### Functions

| | |
|---|---|
| *load_wine* | Load wine dataset. Multiclass problem. |

### Contents

`evalml.demos.wine.load_wine()`

Load wine dataset. Multiclass problem.

**Returns** X and y

**Return type** (pd.Dataframe, pd.Series)

### Package Contents

### Functions

| | |
|---|---|
| *load_breast_cancer* | Load breast cancer dataset. Binary classification problem. |
| *load_churn* | Load churn dataset, which can be used for binary classification problems. |
| *load_diabetes* | Load diabetes dataset. Used for regression problem. |
| *load_fraud* | Load credit card fraud dataset. |
| *load_weather* | Load the Australian daily-min-termperatures weather dataset. |
| *load_wine* | Load wine dataset. Multiclass problem. |

### Contents

`evalml.demos.load_breast_cancer()`

Load breast cancer dataset. Binary classification problem.

**Returns** X and y

**Return type** (pd.Dataframe, pd.Series)

`evalml.demos.load_churn(n_rows=None, verbose=True)`

Load churn dataset, which can be used for binary classification problems.

**Parameters**

- **n_rows** (*int*) – Number of rows from the dataset to return
- **verbose** (*bool*) – Whether to print information about features and labels

**Returns** X and y

**Return type** (pd.Dataframe, pd.Series)

`evalml.demos.load_diabetes()`

Load diabetes dataset. Used for regression problem.

**Returns** X and y

**Return type** (pd.Dataframe, pd.Series)

`evalml.demos.load_fraud(n_rows=None, verbose=True)`

Load credit card fraud dataset.

The fraud dataset can be used for binary classification problems.

**Parameters**

- **n_rows** (*int*) – Number of rows from the dataset to return
- **verbose** (*bool*) – Whether to print information about features and labels

**Returns** X and y

**Return type** (pd.Dataframe, pd.Series)

`evalml.demos.load_weather()`

Load the Australian daily-min-termperatures weather dataset.

**Returns** X and y

**Return type** (pd.Dataframe, pd.Series)

`evalml.demos.load_wine()`

Load wine dataset. Multiclass problem.

**Returns** X and y

**Return type** (pd.Dataframe, pd.Series)

## Exceptions

Exceptions used in EvalML.

## Submodules

### exceptions

Exceptions used in EvalML.

## Module Contents

## Classes Summary

| | |
|---|---|
| *PartialDependenceErrorCode* | Enum identifying the type of error encountered in partial dependence. |
| *PipelineErrorCodeEnum* | Enum identifying the type of error encountered while applying a pipeline. |
| *ValidationErrorCode* | Enum identifying the type of error encountered in hold-out validation. |

## Exceptions Summary

## Contents

**exception** evalml.exceptions.exceptions.**AutoMLSearchException**

Exception raised when all pipelines in an automl batch return a score of NaN for the primary objective.

**exception** evalml.exceptions.exceptions.**ComponentNotYetFittedError**

An exception to be raised when predict/predict_proba/transform is called on a component without fitting first.

**exception** evalml.exceptions.exceptions.**DataCheckInitError**

Exception raised when a data check can't initialize with the parameters given.

**exception** evalml.exceptions.exceptions.**MethodPropertyNotFoundError**

Exception to raise when a class is does not have an expected method or property.

**exception** evalml.exceptions.exceptions.**MissingComponentError**

An exception raised when a component is not found in all_components().

**exception** evalml.exceptions.exceptions.**NoPositiveLabelException**

Exception when a particular classification label for the 'positive' class cannot be found in the column index or unique values.

**exception** evalml.exceptions.exceptions.**NullsInColumnWarning**

Warning thrown when there are null values in the column of interest.

**exception** evalml.exceptions.exceptions.**ObjectiveCreationError**

Exception when get_objective tries to instantiate an objective and required args are not provided.

**exception** evalml.exceptions.exceptions.**ObjectiveNotFoundError**

Exception to raise when specified objective does not exist.

**exception** evalml.exceptions.exceptions.**ParameterNotUsedWarning**(*components*)

Warning thrown when a pipeline parameter isn't used in a defined pipeline's component graph during initialization.

**exception** evalml.exceptions.exceptions.**PartialDependenceError**(*message*, *code*)

Exception raised for all errors that partial dependence can raise.

**Parameters**

- **message** (*str*) – descriptive error message
- **code** (PartialDependenceErrorCode) – code for speicific error

**class** evalml.exceptions.exceptions.**PartialDependenceErrorCode**

Enum identifying the type of error encountered in partial dependence.

**Attributes**

| | |
|---|---|
| **ALL_OTHER_ERRORS** | all_other_errors |
| **COMPUTED_PERCENTILES_TOO_CLOSE** | computed_percentiles_too_close |
| **FEATURE_IS_ALL_NANS** | feature_is_all_nans |
| **FEATURE_IS_MOSTLY_ONE_VALUE** | feature_is_mostly_one_value |
| **FEATURES_ARGUMENT_INCORRECT_TYPES** | features_argument_incorrect_types |
| **ICE_PLOT_REQUESTED_FOR_TWO_WAY_PLOT** | ice_plot_requested_for_two_way_partial_dependence_plot |
| **INVALID_CLASS_LABEL** | invalid_class_label_requested_for_plot |
| **INVALID_FEATURE_TYPE** | invalid_feature_type |
| **PIPELINE_IS_BASELINE** | pipeline_is_baseline |
| **TOO_MANY_FEATURES** | too_many_features |
| **TWO_WAY_REQUESTED_FOR_DATES** | two_way_requested_for_dates |
| **UNFITTED_PIPELINE** | unfitted_pipeline |

**Methods**

| | |
|---|---|
| *name* | The name of the Enum member. |
| *value* | The value of the Enum member. |

**name**(*self*)

The name of the Enum member.

**value**(*self*)

The value of the Enum member.

**exception** evalml.exceptions.exceptions.**PipelineError**(*message*, *code*, *details=None*)

Exception raised for errors that can be raised when applying a pipeline.

**Parameters**

- **message** (*str*) – descriptive error message
- **code** (PipelineErrorCodeEnum) – code for specific error
- **details** (*dict*) – additional details for error

**class** evalml.exceptions.exceptions.**PipelineErrorCodeEnum**

Enum identifying the type of error encountered while applying a pipeline.

**Attributes**

| | |
|---|---|
| **PREDICT_INPUT_SCHEMA_UNEQUAL** | predict_input_schema_unequal |

**Methods**

| | |
|---|---|
| *name* | The name of the Enum member. |
| *value* | The value of the Enum member. |

**name**(*self*)

The name of the Enum member.

**value**(*self*)

The value of the Enum member.

**exception** evalml.exceptions.exceptions.**PipelineNotFoundError**

An exception raised when a particular pipeline is not found in automl search results.

**exception** evalml.exceptions.exceptions.**PipelineNotYetFittedError**

An exception to be raised when predict/predict_proba/transform is called on a pipeline without fitting first.

**exception** evalml.exceptions.exceptions.**PipelineScoreError**(*exceptions*, *scored_successfully*)

An exception raised when a pipeline errors while scoring any objective in a list of objectives.

**Parameters**

- **exceptions** (*dict*) – A dictionary mapping an objective name (str) to a tuple of the form (exception, traceback). All of the objectives that errored will be stored here.
- **scored_successfully** (*dict*) – A dictionary mapping an objective name (str) to a score value. All of the objectives that did not error will be stored here.

**class** evalml.exceptions.exceptions.**ValidationErrorCode**

Enum identifying the type of error encountered in holdout validation.

**Attributes**

| | |
|---|---|
| **INVALID_HOLDOUT_GAP_SEPARATION** | invalid_holdout_gap_separation |
| **INVALID_HOLDOUT_LENGTH** | invalid_holdout_length |

**Methods**

| | |
|---|---|
| *name* | The name of the Enum member. |
| *value* | The value of the Enum member. |

**name**(*self*)

The name of the Enum member.

**value**(*self*)

The value of the Enum member.

### Package Contents

### Classes Summary

| | |
|---|---|
| *PartialDependenceErrorCode* | Enum identifying the type of error encountered in partial dependence. |
| *PipelineErrorCodeEnum* | Enum identifying the type of error encountered while applying a pipeline. |
| *ValidationErrorCode* | Enum identifying the type of error encountered in hold-out validation. |

### Exceptions Summary

### Contents

**exception** evalml.exceptions.**AutoMLSearchException**

Exception raised when all pipelines in an automl batch return a score of NaN for the primary objective.

**exception** evalml.exceptions.**ComponentNotYetFittedError**

An exception to be raised when predict/predict_proba/transform is called on a component without fitting first.

**exception** evalml.exceptions.**DataCheckInitError**

Exception raised when a data check can't initialize with the parameters given.

**exception** evalml.exceptions.**MethodPropertyNotFoundError**

Exception to raise when a class is does not have an expected method or property.

**exception** evalml.exceptions.**MissingComponentError**

An exception raised when a component is not found in all_components().

**exception** evalml.exceptions.**NoPositiveLabelException**

Exception when a particular classification label for the 'positive' class cannot be found in the column index or unique values.

**exception** evalml.exceptions.**NullsInColumnWarning**

Warning thrown when there are null values in the column of interest.

**exception** evalml.exceptions.**ObjectiveCreationError**

Exception when get_objective tries to instantiate an objective and required args are not provided.

**exception** evalml.exceptions.**ObjectiveNotFoundError**

Exception to raise when specified objective does not exist.

**exception** evalml.exceptions.**ParameterNotUsedWarning**(*components*)

Warning thrown when a pipeline parameter isn't used in a defined pipeline's component graph during initialization.

**exception** evalml.exceptions.**PartialDependenceError**(*message*, *code*)

Exception raised for all errors that partial dependence can raise.

**Parameters**

- **message** (*str*) – descriptive error message
- **code** (PartialDependenceErrorCode) – code for speicific error

**class** evalml.exceptions.**PartialDependenceErrorCode**

Enum identifying the type of error encountered in partial dependence.

**Attributes**

| | |
|---|---|
| **ALL_OTHER_ERRORS** | all_other_errors |
| **COMPUTED_PERCENTILES_TOO_CLOSE** | computed_percentiles_too_close |
| **FEATURE_IS_ALL_NANS** | feature_is_all_nans |
| **FEATURE_IS_MOSTLY_ONE_VALUE** | feature_is_mostly_one_value |
| **FEATURES_ARGUMENT_INCORRECT_TYPES** | features_argument_incorrect_types |
| **ICE_PLOT_REQUESTED_FOR_TWO_WAY_PLOT** | ice_plot_requested_for_two_way_partial_dependence_plot |
| **INVALID_CLASS_LABEL** | invalid_class_label_requested_for_plot |
| **INVALID_FEATURE_TYPE** | invalid_feature_type |
| **PIPELINE_IS_BASELINE** | pipeline_is_baseline |
| **TOO_MANY_FEATURES** | too_many_features |
| **TWO_WAY_REQUESTED_FOR_DATES** | two_way_requested_for_dates |
| **UNFITTED_PIPELINE** | unfitted_pipeline |

**Methods**

| | |
|---|---|
| *name* | The name of the Enum member. |
| *value* | The value of the Enum member. |

**name**(*self*)

The name of the Enum member.

**value**(*self*)

The value of the Enum member.

**exception** evalml.exceptions.**PipelineError**(*message*, *code*, *details=None*)

Exception raised for errors that can be raised when applying a pipeline.

**Parameters**

- **message** (*str*) – descriptive error message
- **code** (PipelineErrorCodeEnum) – code for specific error
- **details** (*dict*) – additional details for error

**class** evalml.exceptions.**PipelineErrorCodeEnum**

Enum identifying the type of error encountered while applying a pipeline.

**Attributes**

| | |
|---|---|
| **PREDICT_INPUT_SCHEMA_UNEQUAL** | predict_input_schema_unequal |

**Methods**

| | |
|---|---|
| *name* | The name of the Enum member. |
| *value* | The value of the Enum member. |

**name**(*self*)

The name of the Enum member.

**value**(*self*)

The value of the Enum member.

**exception** evalml.exceptions.**PipelineNotFoundError**

An exception raised when a particular pipeline is not found in automl search results.

**exception** evalml.exceptions.**PipelineNotYetFittedError**

An exception to be raised when predict/predict_proba/transform is called on a pipeline without fitting first.

**exception** evalml.exceptions.**PipelineScoreError**(*exceptions*, *scored_successfully*)

An exception raised when a pipeline errors while scoring any objective in a list of objectives.

**Parameters**

- **exceptions** (*dict*) – A dictionary mapping an objective name (str) to a tuple of the form (exception, traceback). All of the objectives that errored will be stored here.
- **scored_successfully** (*dict*) – A dictionary mapping an objective name (str) to a score value. All of the objectives that did not error will be stored here.

**class** evalml.exceptions.**ValidationErrorCode**

Enum identifying the type of error encountered in holdout validation.

**Attributes**

| | |
|---|---|
| **INVALID_HOLDOUT_GAP_SEPARATION** | invalid_holdout_gap_separation |
| **INVALID_HOLDOUT_LENGTH** | invalid_holdout_length |

**Methods**

| | |
|---|---|
| *name* | The name of the Enum member. |
| *value* | The value of the Enum member. |

**name**(*self*)

The name of the Enum member.

**value**(*self*)

The value of the Enum member.

### Model Family

Family of machine learning models.

### Submodules

### model_family

Enum for family of machine learning models.

### Module Contents

### Classes Summary

| | |
|---|---|
| *ModelFamily* | Enum for family of machine learning models. |

### Contents

**class** evalml.model_family.model_family.**ModelFamily**

Enum for family of machine learning models.

**Attributes**

| | |
|---|---|
| **ARIMA** | ARIMA model family. |
| **BASELINE** | Baseline model family. |
| **CATBOOST** | CatBoost model family. |
| **DECISION_TREE** | Decision Tree model family. |
| **ENSEMBLE** | Ensemble model family. |
| **EXPONENTIAL_SMOOTHING** | Exponential Smoothing model family. |
| **EXTRA_TREES** | Extra Trees model family. |
| **K_NEIGHBORS** | K Nearest Neighbors model family. |
| **LIGHTGBM** | LightGBM model family. |
| **LINEAR_MODEL** | Linear model family. |
| **NONE** | None |
| **PROPHET** | Prophet model family. |
| **RANDOM_FOREST** | Random Forest model family. |
| **SVM** | SVM model family. |
| **VOWPAL_WABBIT** | Vowpal Wabbit model family. |
| **XGBOOST** | XGBoost model family. |

**Methods**

| | |
|---|---|
| *is_tree_estimator* | Checks whether the estimator's model family uses trees. |
| *name* | The name of the Enum member. |
| *value* | The value of the Enum member. |

**is_tree_estimator**(*self*)

Checks whether the estimator's model family uses trees.

**name**(*self*)

The name of the Enum member.

**value**(*self*)

The value of the Enum member.

### utils

Utility methods for EvalML's model families.

### Module Contents

### Functions

| | |
|---|---|
| *handle_model_family* | Handles model_family by either returning the ModelFamily or converting from a string. |

### Contents

evalml.model_family.utils.**handle_model_family**(*model_family*)

Handles model_family by either returning the ModelFamily or converting from a string.

**Parameters** **model_family** (*str or ModelFamily*) – Model type that needs to be handled.

**Returns** ModelFamily

**Raises**

- **KeyError** – If input is not a valid model family.
- **ValueError** – If input is not a string or ModelFamily object.

### Package Contents

### Classes Summary

| | |
|---|---|
| *ModelFamily* | Enum for family of machine learning models. |

### Functions

| | |
|---|---|
| *handle_model_family* | Handles model_family by either returning the ModelFamily or converting from a string. |

### Contents

evalml.model_family.**handle_model_family**(*model_family*)

Handles model_family by either returning the ModelFamily or converting from a string.

**Parameters** **model_family** (*str or* ModelFamily) – Model type that needs to be handled.

**Returns** ModelFamily

**Raises**

- **KeyError** – If input is not a valid model family.
- **ValueError** – If input is not a string or ModelFamily object.

**class** evalml.model_family.**ModelFamily**

Enum for family of machine learning models.

**Attributes**

| | |
|---|---|
| **ARIMA** | ARIMA model family. |
| **BASELINE** | Baseline model family. |
| **CATBOOST** | CatBoost model family. |
| **DECISION_TREE** | Decision Tree model family. |
| **ENSEMBLE** | Ensemble model family. |
| **EXPONENTIAL_SMOOTHING** | Exponential Smoothing model family. |
| **EXTRA_TREES** | Extra Trees model family. |
| **K_NEIGHBORS** | K Nearest Neighbors model family. |
| **LIGHTGBM** | LightGBM model family. |
| **LINEAR_MODEL** | Linear model family. |
| **NONE** | None |
| **PROPHET** | Prophet model family. |
| **RANDOM_FOREST** | Random Forest model family. |
| **SVM** | SVM model family. |
| **VOWPAL_WABBIT** | Vowpal Wabbit model family. |
| **XGBOOST** | XGBoost model family. |

**Methods**

| | |
|---|---|
| *is_tree_estimator* | Checks whether the estimator's model family uses trees. |
| *name* | The name of the Enum member. |
| *value* | The value of the Enum member. |

**is_tree_estimator**(*self*)

Checks whether the estimator's model family uses trees.

**name**(*self*)

The name of the Enum member.

**value**(*self*)

The value of the Enum member.

### Model Understanding

Model understanding tools.

### Subpackages

### prediction_explanations

Prediction explanation tools.

### Submodules

### explainers

Prediction explanation tools.

### Module Contents

### Classes Summary

| | |
|---|---|
| *ExplainPredictionsStage* | Enum for prediction stage. |

### Functions

| | |
|---|---|
| *abs_error* | Computes the absolute error per data point for regression problems. |
| *cross_entropy* | Computes Cross Entropy Loss per data point for classification problems. |
| *explain_predictions* | Creates a report summarizing the top contributing features for each data point in the input features. |
| *explain_predictions_best_worst* | Creates a report summarizing the top contributing features for the best and worst points in the dataset as measured by error to true labels. |

### Attributes Summary

| |
|---|
| *DEFAULT_METRICS* |

### Contents

`evalml.model_understanding.prediction_explanations.explainers.abs_error(y_true, y_pred)`

Computes the absolute error per data point for regression problems.

**Parameters**

- **y_true** (*pd.Series*) – True labels.
- **y_pred** (*pd.Series*) – Predicted values.

**Returns** np.ndarray

`evalml.model_understanding.prediction_explanations.explainers.cross_entropy(y_true, y_pred_proba)`

Computes Cross Entropy Loss per data point for classification problems.

**Parameters**

- **y_true** (*pd.Series*) – True labels encoded as ints.
- **y_pred_proba** (*pd.DataFrame*) – Predicted probabilities. One column per class.

**Returns** np.ndarray

`evalml.model_understanding.prediction_explanations.explainers.DEFAULT_METRICS`

`evalml.model_understanding.prediction_explanations.explainers.explain_predictions(pipeline, input_features, y, indices_to_explain, top_k_features=3, include_explainer_values=Fa include_expected_value=Fals output_format='text', training_data=None, training_target=None, algorithm='shap')`

Creates a report summarizing the top contributing features for each data point in the input features.

XGBoost models and CatBoost multiclass classifiers are not currently supported with the SHAP algorithm. To explain XGBoost model predictions, use the LIME algorithm. The LIME algorithm does not currently support any CatBoost models. For Stacked Ensemble models, the SHAP value for each input pipeline's predict function into the metalearner is used.

**Parameters**

- **pipeline** (*PipelineBase*) – Fitted pipeline whose predictions we want to explain with SHAP or LIME.
- **input_features** (*pd.DataFrame*) – Dataframe of input data to evaluate the pipeline on.
- **y** (*pd.Series*) – Labels for the input data.

- **indices_to_explain** (*list[int]*) – List of integer indices to explain.
- **top_k_features** (*int*) – How many of the highest/lowest contributing feature to include in the table for each data point. Default is 3.
- **include_explainer_values** (*bool*) – Whether explainer (SHAP or LIME) values should be included in the table. Default is False.
- **include_expected_value** (*bool*) – Whether the expected value should be included in the table. Default is False.
- **output_format** (*str*) – Either "text", "dict", or "dataframe". Default is "text".
- **training_data** (*pd.DataFrame, np.ndarray*) – Data the pipeline was trained on. Required and only used for time series pipelines.
- **training_target** (*pd.Series, np.ndarray*) – Targets used to train the pipeline. Required and only used for time series pipelines.
- **algorithm** (*str*) – Algorithm to use while generating top contributing features, one of "shap" or "lime". Defaults to "shap".

**Returns**

**A report explaining the top contributing features to each prediction for each row of input_features.** The report will include the feature names, prediction contribution, and explainer value (optional).

**Return type** str, dict, or pd.DataFrame

**Raises**

- **ValueError** – if input_features is empty.
- **ValueError** – if an output_format outside of "text", "dict" or "dataframe is provided.
- **ValueError** – if the requested index falls outside the input_feature's boundaries.

`evalml.model_understanding.prediction_explanations.explainers.`**`explain_predictions_best_worst`**(*pipeline, input_features, y_true, num_to_expla top_k_feature in-clude_explain met-ric=None, out-put_format=' call-back=None, train-ing_data=No train-ing_target=N al-go-rithm='shap'*)

Creates a report summarizing the top contributing features for the best and worst points in the dataset as measured by error to true labels.

XGBoost models and CatBoost multiclass classifiers are not currently supported with the SHAP algorithm. To explain XGBoost model predictions, use the LIME algorithm. The LIME algorithm does not currently support any CatBoost models. For Stacked Ensemble models, the SHAP value for each input pipeline's predict function into the metalearner is used.

**Parameters**

- **pipeline** (*PipelineBase*) – Fitted pipeline whose predictions we want to explain with SHAP or LIME.
- **input_features** (*pd.DataFrame*) – Input data to evaluate the pipeline on.
- **y_true** (*pd.Series*) – True labels for the input data.
- **num_to_explain** (*int*) – How many of the best, worst, random data points to explain.
- **top_k_features** (*int*) – How many of the highest/lowest contributing feature to include in the table for each data point.
- **include_explainer_values** (*bool*) – Whether explainer (SHAP or LIME) values should be included in the table. Default is False.
- **metric** (*callable*) – The metric used to identify the best and worst points in the dataset. Function must accept the true labels and predicted value or probabilities as the only arguments and lower values must be better. By default, this will be the absolute error for regression problems and cross entropy loss for classification problems.
- **output_format** (*str*) – Either "text" or "dict". Default is "text".
- **callback** (*callable*) – Function to be called with incremental updates. Has the following parameters: - progress_stage: stage of computation - time_elapsed: total time in seconds that has elapsed since start of call
- **training_data** (*pd.DataFrame, np.ndarray*) – Data the pipeline was trained on. Required and only used for time series pipelines.
- **training_target** (*pd.Series, np.ndarray*) – Targets used to train the pipeline. Required and only used for time series pipelines.
- **algorithm** (*str*) – Algorithm to use while generating top contributing features, one of "shap" or "lime". Defaults to "shap".

**Returns**

**A report explaining the top contributing features for the best/worst predictions in the input_features.** For each of the best/worst rows of input_features, the predicted values, true labels, metric value, feature names, prediction contribution, and explainer value (optional) will be listed.

**Return type** str, dict, or pd.DataFrame

**Raises**

- **ValueError** – If input_features does not have more than twice the requested features to explain.
- **ValueError** – If y_true and input_features have mismatched lengths.
- **ValueError** – If an output_format outside of "text", "dict" or "dataframe is provided.
- **PipelineScoreError** – If the pipeline errors out while scoring.

**class** evalml.model_understanding.prediction_explanations.explainers.**ExplainPredictionsStage**

Enum for prediction stage.

**Attributes**

| | |
|---|---|
| **COMPUTE_EXPLAINER_VALUES_STAGE** | compute_explainer_value_stage |
| **COMPUTE_FEATURE_STAGE** | compute_feature_stage |
| **DONE** | done |
| **PREDICT_STAGE** | predict_stage |
| **PREPROCESSING_STAGE** | preprocessing_stage |

**Methods**

| | |
|---|---|
| *name* | The name of the Enum member. |
| *value* | The value of the Enum member. |

**name**(*self*)

The name of the Enum member.

**value**(*self*)

The value of the Enum member.

## Package Contents

## Functions

| | |
|---|---|
| *explain_predictions* | Creates a report summarizing the top contributing features for each data point in the input features. |
| *explain_predictions_best_worst* | Creates a report summarizing the top contributing features for the best and worst points in the dataset as measured by error to true labels. |

## Contents

evalml.model_understanding.prediction_explanations.**explain_predictions**(*pipeline*, *input_features*, *y*, *indices_to_explain*, *top_k_features=3*, *include_explainer_values=False*, *include_expected_value=False*, *output_format='text'*, *training_data=None*, *training_target=None*, *algorithm='shap'*)

Creates a report summarizing the top contributing features for each data point in the input features.

XGBoost models and CatBoost multiclass classifiers are not currently supported with the SHAP algorithm. To explain XGBoost model predictions, use the LIME algorithm. The LIME algorithm does not currently support

any CatBoost models. For Stacked Ensemble models, the SHAP value for each input pipeline's predict function into the metalearner is used.

**Parameters**

- **pipeline** (*PipelineBase*) – Fitted pipeline whose predictions we want to explain with SHAP or LIME.
- **input_features** (*pd.DataFrame*) – Dataframe of input data to evaluate the pipeline on.
- **y** (*pd.Series*) – Labels for the input data.
- **indices_to_explain** (*list[int]*) – List of integer indices to explain.
- **top_k_features** (*int*) – How many of the highest/lowest contributing feature to include in the table for each data point. Default is 3.
- **include_explainer_values** (*bool*) – Whether explainer (SHAP or LIME) values should be included in the table. Default is False.
- **include_expected_value** (*bool*) – Whether the expected value should be included in the table. Default is False.
- **output_format** (*str*) – Either "text", "dict", or "dataframe". Default is "text".
- **training_data** (*pd.DataFrame, np.ndarray*) – Data the pipeline was trained on. Required and only used for time series pipelines.
- **training_target** (*pd.Series, np.ndarray*) – Targets used to train the pipeline. Required and only used for time series pipelines.
- **algorithm** (*str*) – Algorithm to use while generating top contributing features, one of "shap" or "lime". Defaults to "shap".

**Returns**

**A report explaining the top contributing features to each prediction for each row of input_features.** The report will include the feature names, prediction contribution, and explainer value (optional).

**Return type** str, dict, or pd.DataFrame

**Raises**

- **ValueError** – if input_features is empty.
- **ValueError** – if an output_format outside of "text", "dict" or "dataframe is provided.
- **ValueError** – if the requested index falls outside the input_feature's boundaries.

evalml.model_understanding.prediction_explanations.**explain_predictions_best_worst**(*pipeline*, *input_features*, *y_true*, *num_to_explain=5*, *top_k_features=3*, *include_explainer_values=Fa*
*metric=None*, *output_format='text'*, *callback=None*, *training_data=None*, *training_target=None*, *algorithm='shap'*)

Creates a report summarizing the top contributing features for the best and worst points in the dataset as measured by error to true labels.

XGBoost models and CatBoost multiclass classifiers are not currently supported with the SHAP algorithm. To explain XGBoost model predictions, use the LIME algorithm. The LIME algorithm does not currently support any CatBoost models. For Stacked Ensemble models, the SHAP value for each input pipeline's predict function into the metalearner is used.

**Parameters**

- **pipeline** (*PipelineBase*) – Fitted pipeline whose predictions we want to explain with SHAP or LIME.
- **input_features** (*pd.DataFrame*) – Input data to evaluate the pipeline on.
- **y_true** (*pd.Series*) – True labels for the input data.
- **num_to_explain** (*int*) – How many of the best, worst, random data points to explain.
- **top_k_features** (*int*) – How many of the highest/lowest contributing feature to include in the table for each data point.
- **include_explainer_values** (*bool*) – Whether explainer (SHAP or LIME) values should be included in the table. Default is False.
- **metric** (*callable*) – The metric used to identify the best and worst points in the dataset. Function must accept the true labels and predicted value or probabilities as the only arguments and lower values must be better. By default, this will be the absolute error for regression problems and cross entropy loss for classification problems.
- **output_format** (*str*) – Either "text" or "dict". Default is "text".
- **callback** (*callable*) – Function to be called with incremental updates. Has the following parameters: - progress_stage: stage of computation - time_elapsed: total time in seconds that has elapsed since start of call
- **training_data** (*pd.DataFrame, np.ndarray*) – Data the pipeline was trained on. Required and only used for time series pipelines.

- **training_target** (*pd.Series, np.ndarray*) – Targets used to train the pipeline. Required and only used for time series pipelines.
- **algorithm** (*str*) – Algorithm to use while generating top contributing features, one of "shap" or "lime". Defaults to "shap".

**Returns**

**A report explaining the top contributing features for the best/worst predictions in the input_features.** For each of the best/worst rows of input_features, the predicted values, true labels, metric value, feature names, prediction contribution, and explainer value (optional) will be listed.

**Return type** str, dict, or pd.DataFrame

**Raises**

- **ValueError** – If input_features does not have more than twice the requested features to explain.
- **ValueError** – If y_true and input_features have mismatched lengths.
- **ValueError** – If an output_format outside of "text", "dict" or "dataframe is provided.
- **PipelineScoreError** – If the pipeline errors out while scoring.

## Submodules

### decision_boundary

Model Understanding for decision boundary on Binary Classification problems.

### Module Contents

### Functions

| | |
|---|---|
| *find_confusion_matrix_per_thresholds* | Gets the confusion matrix and histogram bins for each threshold as well as the best threshold per objective. Only works with Binary Classification Pipelines. |

### Contents

evalml.model_understanding.decision_boundary.**find_confusion_matrix_per_thresholds**(*pipeline*, *X*, *y*, *n_bins=None*, *top_k=5*, *to_json=False*)

Gets the confusion matrix and histogram bins for each threshold as well as the best threshold per objective. Only works with Binary Classification Pipelines.

**Parameters**

- **pipeline** (*PipelineBase*) – A fitted Binary Classification Pipeline to get the confusion matrix with.
- **X** (*pd.DataFrame*) – The input features.

- **y** (*pd.Series*) – The input target.
- **n_bins** (*int*) – The number of bins to use to calculate the threshold values. Defaults to None, which will default to using Freedman-Diaconis rule.
- **top_k** (*int*) – The maximum number of row indices per bin to include as samples. -1 includes all row indices that fall between the bins. Defaults to 5.
- **to_json** (*bool*) – Whether or not to return a json output. If False, returns the (DataFrame, dict) tuple, otherwise returns a json.

**Returns**

**The dataframe has the actual positive histogram, actual negative histogram,** the confusion matrix, and a sample of rows that fall in the bin, all for each threshold value. The threshold value, represented through the dataframe index, represents the cutoff threshold at that value. The dictionary contains the ideal threshold and score per objective, keyed by objective name. If json, returns the info for both the dataframe and dictionary as a json output.

**Return type** (tuple(pd.DataFrame, dict)), json)

**Raises** **ValueError** – If the pipeline isn't a binary classification pipeline or isn't yet fitted on data.

## feature_explanations

Human Readable Pipeline Explanations.

### Module Contents

### Functions

| | |
|---|---|
| *get_influential_features* | Finds the most influential features as well as any detrimental features from a dataframe of feature importances. |
| *readable_explanation* | Outputs a human-readable explanation of trained pipeline behavior. |

### Contents

evalml.model_understanding.feature_explanations.**get_influential_features**(*imp_df, max_features=5, min_importance_threshold=0.05, linear_importance=False*)

Finds the most influential features as well as any detrimental features from a dataframe of feature importances.

**Parameters**

- **imp_df** (*pd.DataFrame*) – DataFrame containing feature names and associated importances.
- **max_features** (*int*) – The maximum number of features to include in an explanation. Defaults to 5.
- **min_importance_threshold** (*float*) – The minimum percent of total importance a single feature can have to be considered important. Defaults to 0.05.

- **linear_importance** (*bool*) – When True, negative feature importances are not considered detrimental. Defaults to False.

**Returns** Lists of feature names corresponding to heavily influential, somewhat influential, and detrimental features, respectively.

**Return type** (list, list, list)

`evalml.model_understanding.feature_explanations.readable_explanation(pipeline, X=None, y=None, importance_method='permutation', max_features=5, min_importance_threshold=0.05, objective='auto')`

Outputs a human-readable explanation of trained pipeline behavior.

**Parameters**

- **pipeline** (*PipelineBase*) – The pipeline to explain.
- **X** (*pd.DataFrame*) – If importance_method is permutation, the holdout X data to compute importance with. Ignored otherwise.
- **y** (*pd.Series*) – The holdout y data, used to obtain the name of the target class. If importance_method is permutation, used to compute importance with.
- **importance_method** (*str*) – The method of determining feature importance. One of ["permutation", "feature"]. Defaults to "permutation".
- **max_features** (*int*) – The maximum number of influential features to include in an explanation. This does not affect the number of detrimental features reported. Defaults to 5.
- **min_importance_threshold** (*float*) – The minimum percent of total importance a single feature can have to be considered important. Defaults to 0.05.
- **objective** (*str, ObjectiveBase*) – If importance_method is permutation, the objective to compute importance with. Ignored otherwise, defaults to "auto".

**Raises** **ValueError** – if any arguments passed in are invalid or the pipeline is not fitted.

### force_plots

Force plots.

### Module Contents

### Functions

| | |
|---|---|
| *force_plot* | Function to generate the data required to build a force plot. |
| *graph_force_plot* | Function to generate force plots for the desired rows of the training data. |

### Contents

`evalml.model_understanding.force_plots.force_plot(pipeline, rows_to_explain, training_data, y)`

Function to generate the data required to build a force plot.

**Parameters**

- **pipeline** (*PipelineBase*) – The pipeline to generate the force plot for.
- **rows_to_explain** (*list[int]*) – A list of the indices of the training_data to explain.
- **training_data** (*pandas.DataFrame*) – The data used to train the pipeline.
- **y** (*pandas.Series*) – The target data.

**Returns**

**list of dictionaries where each dict contains force plot data. Each dictionary** entry represents the explanations for a single row.

**For single row binary force plots:**

**[{'malignant': {'expected_value': 0.37,** 'feature_names': ['worst concave points', 'worst perimeter', 'worst radius'], 'shap_values': [0.09, 0.09, 0.08], 'plot': AdditiveForceVisualizer}]

**For two row binary force plots:**

**[{'malignant': {'expected_value': 0.37,** 'feature_names': ['worst concave points', 'worst perimeter', 'worst radius'], 'shap_values': [0.09, 0.09, 0.08], 'plot': AdditiveForceVisualizer},

**{'malignant': {'expected_value': 0.29,** 'feature_names': ['worst concave points', 'worst perimeter', 'worst radius'], 'shap_values': [0.05, 0.03, 0.02], 'plot': AdditiveForceVisualizer}]

**Return type** list[dict]

**Raises**

- **TypeError** – If rows_to_explain is not a list.
- **TypeError** – If all values in rows_to_explain aren't integers.

`evalml.model_understanding.force_plots.graph_force_plot(pipeline, rows_to_explain, training_data, y, matplotlib=False)`

Function to generate force plots for the desired rows of the training data.

**Parameters**

- **pipeline** (*PipelineBase*) – The pipeline to generate the force plot for.
- **rows_to_explain** (*list[int]*) – A list of the indices indicating which of the rows of the training_data to explain.
- **training_data** (*pandas.DataFrame*) – The data used to train the pipeline.
- **y** (*pandas.Series*) – The target data for the pipeline.
- **matplotlib** (*bool*) – flag to display the force plot using matplotlib (outside of jupyter) Defaults to False.

**Returns**

**The same as force_plot(), but with an additional** key in each dictionary for the plot.

**Return type** list[dict[shap.AdditiveForceVisualizer]]

### metrics

Standard metrics used for model understanding.

### Module Contents

### Functions

| | |
|---|---|
| *confusion_matrix* | Confusion matrix for binary and multiclass classification. |
| *graph_confusion_matrix* | Generate and display a confusion matrix plot. |
| *graph_precision_recall_curve* | Generate and display a precision-recall plot. |
| *graph_roc_curve* | Generate and display a Receiver Operating Characteristic (ROC) plot for binary and multiclass classification problems. |
| *normalize_confusion_matrix* | Normalizes a confusion matrix. |
| *precision_recall_curve* | Given labels and binary classifier predicted probabilities, compute and return the data representing a precision-recall curve. |
| *roc_curve* | Given labels and classifier predicted probabilities, compute and return the data representing a Receiver Operating Characteristic (ROC) curve. Works with binary or multiclass problems. |

### Contents

evalml.model_understanding.metrics.**confusion_matrix**(*y_true*, *y_predicted*, *normalize_method='true'*)

Confusion matrix for binary and multiclass classification.

**Parameters**

- **y_true** (*pd.Series or np.ndarray*) – True binary labels.
- **y_predicted** (*pd.Series or np.ndarray*) – Predictions from a binary classifier.
- **normalize_method** (*{'true', 'pred', 'all', None}*) – Normalization method to use, if not None. Supported options are: 'true' to normalize by row, 'pred' to normalize by column, or 'all' to normalize by all values. Defaults to 'true'.

**Returns** Confusion matrix. The column header represents the predicted labels while row header represents the actual labels.

**Return type** pd.DataFrame

evalml.model_understanding.metrics.**graph_confusion_matrix**(*y_true*, *y_pred*, *normalize_method='true'*, *title_addition=None*)

Generate and display a confusion matrix plot.

If *normalize_method* is set, hover text will show raw count, otherwise hover text will show count normalized with method 'true'.

**Parameters**

- **y_true** (*pd.Series or np.ndarray*) – True binary labels.
- **y_pred** (*pd.Series or np.ndarray*) – Predictions from a binary classifier.
- **normalize_method** (*{'true', 'pred', 'all', None}*) – Normalization method to use, if not None. Supported options are: 'true' to normalize by row, 'pred' to normalize by column, or 'all' to normalize by all values. Defaults to 'true'.
- **title_addition** (*str*) – If not None, append to plot title. Defaults to None.

**Returns** plotly.Figure representing the confusion matrix plot generated.

evalml.model_understanding.metrics.**graph_precision_recall_curve**(*y_true*, *y_pred_proba*, *title_addition=None*)

Generate and display a precision-recall plot.

**Parameters**

- **y_true** (*pd.Series or np.ndarray*) – True binary labels.
- **y_pred_proba** (*pd.Series or np.ndarray*) – Predictions from a binary classifier, before thresholding has been applied. Note this should be the predicted probability for the "true" label.
- **title_addition** (*str or None*) – If not None, append to plot title. Defaults to None.

**Returns** plotly.Figure representing the precision-recall plot generated

evalml.model_understanding.metrics.**graph_roc_curve**(*y_true*, *y_pred_proba*, *custom_class_names=None*, *title_addition=None*)

Generate and display a Receiver Operating Characteristic (ROC) plot for binary and multiclass classification problems.

**Parameters**

- **y_true** (*pd.Series or np.ndarray*) – True labels.
- **y_pred_proba** (*pd.Series or np.ndarray*) – Predictions from a classifier, before thresholding has been applied. Note this should a one dimensional array with the predicted probability for the "true" label in the binary case.
- **custom_class_names** (*list or None*) – If not None, custom labels for classes. Defaults to None.
- **title_addition** (*str or None*) – if not None, append to plot title. Defaults to None.

**Returns** plotly.Figure representing the ROC plot generated

**Raises** **ValueError** – If the number of custom class names does not match number of classes in the input data.

evalml.model_understanding.metrics.**normalize_confusion_matrix**(*conf_mat*, *normalize_method='true'*)

Normalizes a confusion matrix.

**Parameters**

- **conf_mat** (*pd.DataFrame or np.ndarray*) – Confusion matrix to normalize.
- **normalize_method** (*{'true', 'pred', 'all'}*) – Normalization method. Supported options are: 'true' to normalize by row, 'pred' to normalize by column, or 'all' to normalize by all values. Defaults to 'true'.

**Returns** normalized version of the input confusion matrix. The column header represents the predicted labels while row header represents the actual labels.

**Return type** pd.DataFrame

**Raises** `ValueError` – If configuration is invalid, or if the sum of a given axis is zero and normalization by axis is specified.

`evalml.model_understanding.metrics.`**`precision_recall_curve`**(*y_true*, *y_pred_proba*, *pos_label_idx=-1*)

Given labels and binary classifier predicted probabilities, compute and return the data representing a precision-recall curve.

**Parameters**

- **y_true** (*pd.Series or np.ndarray*) – True binary labels.
- **y_pred_proba** (*pd.Series or np.ndarray*) – Predictions from a binary classifier, before thresholding has been applied. Note this should be the predicted probability for the "true" label.
- **pos_label_idx** (*int*) – the column index corresponding to the positive class. If predicted probabilities are two-dimensional, this will be used to access the probabilities for the positive class.

**Returns**

Dictionary containing metrics used to generate a precision-recall plot, with the following keys:

- *precision*: Precision values.
- *recall*: Recall values.
- *thresholds*: Threshold values used to produce the precision and recall.
- *auc_score*: The area under the ROC curve.

**Return type** list

**Raises** `NoPositiveLabelException` – If predicted probabilities do not contain a column at the specified label.

`evalml.model_understanding.metrics.`**`roc_curve`**(*y_true*, *y_pred_proba*)

Given labels and classifier predicted probabilities, compute and return the data representing a Receiver Operating Characteristic (ROC) curve. Works with binary or multiclass problems.

**Parameters**

- **y_true** (*pd.Series or np.ndarray*) – True labels.
- **y_pred_proba** (*pd.Series or pd.DataFrame or np.ndarray*) – Predictions from a classifier, before thresholding has been applied.

**Returns**

**A list of dictionaries (with one for each class) is returned. Binary classification problems return a list with one di**

**Each dictionary contains metrics used to generate an ROC plot with the following keys:**

- *fpr_rate*: False positive rate.
- *tpr_rate*: True positive rate.
- *threshold*: Threshold values used to produce each pair of true/false positive rates.

- *auc_score*: The area under the ROC curve.

**Return type** list(dict)

### partial_dependence_functions

Top level functions for running partial dependence.

### Module Contents

### Functions

| | |
|---|---|
| *graph_partial_dependence* | Create an one-way or two-way partial dependence plot. |
| *partial_dependence* | Calculates one or two-way partial dependence. |

### Contents

evalml.model_understanding.partial_dependence_functions.**graph_partial_dependence**(*pipeline*, *X*, *features*, *class_label=None*, *grid_resolution=100*, *kind='average'*)

Create an one-way or two-way partial dependence plot.

Passing a single integer or string as features will create a one-way partial dependence plot with the feature values plotted against the partial dependence. Passing features a tuple of int/strings will create a two-way partial dependence plot with a contour of feature[0] in the y-axis, feature[1] in the x-axis and the partial dependence in the z-axis.

**Parameters**

- **pipeline** (*PipelineBase or subclass*) – Fitted pipeline.
- **X** (*pd.DataFrame, np.ndarray*) – The input data used to generate a grid of values for feature where partial dependence will be calculated at.
- **features** (*int, string, tuple[int or string]*) – The target feature for which to create the partial dependence plot for. If features is an int, it must be the index of the feature to use. If features is a string, it must be a valid column name in X. If features is a tuple of strings, it must contain valid column int/names in X.
- **class_label** (*string, optional*) – Name of class to plot for multiclass problems. If None, will plot the partial dependence for each class. This argument does not change behavior for regression or binary classification pipelines. For binary classification, the partial dependence for the positive label will always be displayed. Defaults to None.
- **grid_resolution** (*int*) – Number of samples of feature(s) for partial dependence plot.
- **kind** (*{'average', 'individual', 'both'}*) – Type of partial dependence to plot. 'average' creates a regular partial dependence (PD) graph, 'individual' creates an individual conditional expectation (ICE) plot, and 'both' creates a single-figure PD and ICE plot. ICE plots can only be shown for one-way partial dependence plots.

**Returns** figure object containing the partial dependence data for plotting

**Return type** plotly.graph_objects.Figure

**Raises**

- **PartialDependenceError** – if a graph is requested for a class name that isn't present in the pipeline.
- **PartialDependenceError** – if an ICE plot is requested for a two-way partial dependence.

evalml.model_understanding.partial_dependence_functions.**partial_dependence**(*pipeline*, *X*, *features*, *percentiles=(0.05, 0.95)*, *grid_resolution=100*, *kind='average'*, *fast_mode=False*, *X_train=None*, *y_train=None*)

Calculates one or two-way partial dependence.

If a single integer or string is given for features, one-way partial dependence is calculated. If a tuple of two integers or strings is given, two-way partial dependence is calculated with the first feature in the y-axis and second feature in the x-axis.

**Parameters**

- **pipeline** (*PipelineBase or subclass*) – Fitted pipeline
- **X** (*pd.DataFrame, np.ndarray*) – The input data used to generate a grid of values for feature where partial dependence will be calculated at
- **features** (*int, string, tuple[int or string]*) – The target feature for which to create the partial dependence plot for. If features is an int, it must be the index of the feature to use. If features is a string, it must be a valid column name in X. If features is a tuple of int/strings, it must contain valid column integers/names in X.
- **percentiles** (*tuple[float]*) – The lower and upper percentile used to create the extreme values for the grid. Must be in [0, 1]. Defaults to (0.05, 0.95).
- **grid_resolution** (*int*) – Number of samples of feature(s) for partial dependence plot. If this value is less than the maximum number of categories present in categorical data within X, it will be set to the max number of categories + 1. Defaults to 100.
- **kind** (*{'average', 'individual', 'both'}*) – The type of predictions to return. 'individual' will return the predictions for all of the points in the grid for each sample in X. 'average' will return the predictions for all of the points in the grid but averaged over all of the samples in X.
- **fast_mode** (*bool, optional*) – Whether or not performance optimizations should be used for partial dependence calculations. Defaults to False. Note that user-specified components may not produce correct partial dependence results, so fast mode should only be used with EvalML-native components. Additionally, some components are not compatible with fast mode; in those cases, an error will be raised indicating that fast mode should not be used.
- **X_train** (*pd.DataFrame, np.ndarray*) – The data that was used to train the original pipeline. Will be used in fast mode to train the cloned pipelines. Defaults to None.
- **y_train** (*pd.Series, np.ndarray*) – The target data that was used to train the original pipeline. Will be used in fast mode to train the cloned pipelines. Defaults to None.

**Returns**

When *kind='average'*: DataFrame with averaged predictions for all points in the grid averaged over all samples of X and the values used to calculate those predictions.

When *kind='individual'*: DataFrame with individual predictions for all points in the grid for each sample of X and the values used to calculate those predictions. If a two-way partial dependence is calculated, then the result is a list of DataFrames with each DataFrame representing one sample's predictions.

When *kind='both'*: A tuple consisting of the averaged predictions (in a DataFrame) over all samples of X and the individual predictions (in a list of DataFrames) for each sample of X.

In the one-way case: The dataframe will contain two columns, "feature_values" (grid points at which the partial dependence was calculated) and "partial_dependence" (the partial dependence at that feature value). For classification problems, there will be a third column called "class_label" (the class label for which the partial dependence was calculated). For binary classification, the partial dependence is only calculated for the "positive" class.

In the two-way case: The data frame will contain grid_resolution number of columns and rows where the index and column headers are the sampled values of the first and second features, respectively, used to make the partial dependence contour. The values of the data frame contain the partial dependence data for each feature value pair.

**Return type** pd.DataFrame, list(pd.DataFrame), or tuple(pd.DataFrame, list(pd.DataFrame))

**Raises**

- `ValueError` – Error during call to scikit-learn's partial dependence method.
- `Exception` – All other errors during calculation.
- `PartialDependenceError` – if the user provides a tuple of not exactly two features.
- `PartialDependenceError` – if the provided pipeline isn't fitted.
- `PartialDependenceError` – if the provided pipeline is a Baseline pipeline.
- `PartialDependenceError` – if any of the features passed in are completely NaN
- `PartialDependenceError` – if any of the features are low-variance. Defined as having one value occurring more than the upper percentile passed by the user. By default 95%.

## permutation_importance

Permutation importance methods.

### Module Contents

### Functions

| | |
|---|---|
| *calculate_permutation_importance* | Calculates permutation importance for features. |
| *calculate_permutation_importance_one_column* | Calculates permutation importance for one column in the original dataframe. |
| *graph_permutation_importance* | Generate a bar graph of the pipeline's permutation importance. |

## Contents

`evalml.model_understanding.permutation_importance.`**`calculate_permutation_importance`**(*pipeline, X, y, objective, n_repeats=5, n_jobs=None, random_seed=0*)

Calculates permutation importance for features.

**Parameters**

- **pipeline** (*PipelineBase or subclass*) – Fitted pipeline.
- **X** (*pd.DataFrame*) – The input data used to score and compute permutation importance.
- **y** (*pd.Series*) – The target data.
- **objective** (*str, ObjectiveBase*) – Objective to score on.
- **n_repeats** (*int*) – Number of times to permute a feature. Defaults to 5.
- **n_jobs** (*int or None*) – Non-negative integer describing level of parallelism used for pipelines. None and 1 are equivalent. If set to -1, all CPUs are used. For n_jobs below -1, (n_cpus + 1 + n_jobs) are used. Defaults to None.
- **random_seed** (*int*) – Seed for the random number generator. Defaults to 0.

**Returns** Mean feature importance scores over a number of shuffles.

**Return type** pd.DataFrame

**Raises** **ValueError** – If objective cannot be used with the given pipeline.

`evalml.model_understanding.permutation_importance.`**`calculate_permutation_importance_one_column`**(*pipeline, X, y, col_name, objective, n_repeats=5, fast=True, precomputed_featur ran- dom_seed=0*

Calculates permutation importance for one column in the original dataframe.

**Parameters**

- **pipeline** (*PipelineBase or subclass*) – Fitted pipeline.
- **X** (*pd.DataFrame*) – The input data used to score and compute permutation importance.
- **y** (*pd.Series*) – The target data.
- **col_name** (*str, int*) – The column in X to calculate permutation importance for.
- **objective** (*str, ObjectiveBase*) – Objective to score on.

- **n_repeats** (*int*) – Number of times to permute a feature. Defaults to 5.
- **fast** (*bool*) – Whether to use the fast method of calculating the permutation importance or not. Defaults to True.
- **precomputed_features** (*pd.DataFrame*) – Precomputed features necessary to calculate permutation importance using the fast method. Defaults to None.
- **random_seed** (*int*) – Seed for the random number generator. Defaults to 0.

**Returns** Mean feature importance scores over a number of shuffles.

**Return type** float

**Raises**

- **ValueError** – If pipeline does not support fast permutation importance calculation.
- **ValueError** – If precomputed_features is None.

evalml.model_understanding.permutation_importance.**graph_permutation_importance**(*pipeline*, *X*, *y*, *objective*, *importance_threshold=0*)

Generate a bar graph of the pipeline's permutation importance.

**Parameters**

- **pipeline** (*PipelineBase or subclass*) – Fitted pipeline.
- **X** (*pd.DataFrame*) – The input data used to score and compute permutation importance.
- **y** (*pd.Series*) – The target data.
- **objective** (*str, ObjectiveBase*) – Objective to score on.
- **importance_threshold** (*float, optional*) – If provided, graph features with a permutation importance whose absolute value is larger than importance_threshold. Defaults to 0.

**Returns** plotly.Figure, a bar graph showing features and their respective permutation importance.

**Raises** **ValueError** – If importance_threshold is not greater than or equal to 0.

### visualizations

Visualization functions for model understanding.

### Module Contents

### Functions

| | |
|---|---|
| *binary_objective_vs_threshold* | Computes objective score as a function of potential binary classification decision thresholds for a fitted binary classification pipeline. |
| *decision_tree_data_from_estimator* | Return data for a fitted tree in a restructured format. |
| *decision_tree_data_from_pipeline* | Return data for a fitted pipeline in a restructured format. |
| *get_linear_coefficients* | Returns a dataframe showing the features with the greatest predictive power for a linear model. |
| *get_prediction_vs_actual_data* | Combines y_true and y_pred into a single dataframe and adds a column for outliers. Used in *graph_prediction_vs_actual()*. |
| *get_prediction_vs_actual_over_time_data* | Get the data needed for the prediction_vs_actual_over_time plot. |
| *graph_binary_objective_vs_threshold* | Generates a plot graphing objective score vs. decision thresholds for a fitted binary classification pipeline. |
| *graph_prediction_vs_actual* | Generate a scatter plot comparing the true and predicted values. Used for regression plotting. |
| *graph_prediction_vs_actual_over_time* | Plot the target values and predictions against time on the x-axis. |
| *graph_t_sne* | Plot high dimensional data into lower dimensional space using t-SNE. |
| *t_sne* | Get the transformed output after fitting X to the embedded space using t-SNE. |
| *visualize_decision_tree* | Generate an image visualizing the decision tree. |

### Contents

evalml.model_understanding.visualizations.**binary_objective_vs_threshold**(*pipeline*, *X*, *y*, *objective*, *steps=100*)

Computes objective score as a function of potential binary classification decision thresholds for a fitted binary classification pipeline.

**Parameters**

- **pipeline** (*BinaryClassificationPipeline obj*) – Fitted binary classification pipeline.
- **X** (*pd.DataFrame*) – The input data used to compute objective score.
- **y** (*pd.Series*) – The target labels.
- **objective** (*ObjectiveBase obj, str*) – Objective used to score.
- **steps** (*int*) – Number of intervals to divide and calculate objective score at.

**Returns** DataFrame with thresholds and the corresponding objective score calculated at each threshold.

**Return type** pd.DataFrame

**Raises**

- **ValueError** – If objective is not a binary classification objective.
- **ValueError** – If objective's *score_needs_proba* is not False.

evalml.model_understanding.visualizations.**decision_tree_data_from_estimator**(*estimator*)

Return data for a fitted tree in a restructured format.

**Parameters** **estimator** (*ComponentBase*) – A fitted DecisionTree-based estimator.

**Returns** An OrderedDict of OrderedDicts describing a tree structure.

**Return type** OrderedDict

**Raises**

- **ValueError** – If estimator is not a decision tree-based estimator.
- **NotFittedError** – If estimator is not yet fitted.

evalml.model_understanding.visualizations.**decision_tree_data_from_pipeline**(*pipeline_*)

Return data for a fitted pipeline in a restructured format.

**Parameters** **pipeline** (*PipelineBase*) – A pipeline with a DecisionTree-based estimator.

**Returns** An OrderedDict of OrderedDicts describing a tree structure.

**Return type** OrderedDict

**Raises**

- **ValueError** – If estimator is not a decision tree-based estimator.
- **NotFittedError** – If estimator is not yet fitted.

evalml.model_understanding.visualizations.**get_linear_coefficients**(*estimator*, *features=None*)

Returns a dataframe showing the features with the greatest predictive power for a linear model.

**Parameters**

- **estimator** (*Estimator*) – Fitted linear model family estimator.
- **features** (*list[str]*) – List of feature names associated with the underlying data.

**Returns** Displaying the features by importance.

**Return type** pd.DataFrame

**Raises**

- **ValueError** – If the model is not a linear model.
- **NotFittedError** – If the model is not yet fitted.

evalml.model_understanding.visualizations.**get_prediction_vs_actual_data**(*y_true*, *y_pred*, *outlier_threshold=None*)

Combines y_true and y_pred into a single dataframe and adds a column for outliers. Used in *graph_prediction_vs_actual()*.

**Parameters**

- **y_true** (*pd.Series, or np.ndarray*) – The real target values of the data
- **y_pred** (*pd.Series, or np.ndarray*) – The predicted values outputted by the regression model.
- **outlier_threshold** (*int, float*) – A positive threshold for what is considered an outlier value. This value is compared to the absolute difference between each value of y_true and y_pred. Values within this threshold will be blue, otherwise they will be yellow. Defaults to None.

**Returns**

- *prediction*: Predicted values from regression model.
- *actual*: Real target values.
- *outlier*: Colors indicating which values are in the threshold for what is considered an outlier value.

**Return type** pd.DataFrame with the following columns

**Raises** `ValueError` – If threshold is not positive.

`evalml.model_understanding.visualizations.get_prediction_vs_actual_over_time_data(pipeline, X, y, X_train, y_train, dates)`

Get the data needed for the prediction_vs_actual_over_time plot.

**Parameters**

- **pipeline** (*TimeSeriesRegressionPipeline*) – Fitted time series regression pipeline.
- **X** (*pd.DataFrame*) – Features used to generate new predictions.
- **y** (*pd.Series*) – Target values to compare predictions against.
- **X_train** (*pd.DataFrame*) – Data the pipeline was trained on.
- **y_train** (*pd.Series*) – Target values for training data.
- **dates** (*pd.Series*) – Dates corresponding to target values and predictions.

**Returns** Predictions vs. time.

**Return type** pd.DataFrame

`evalml.model_understanding.visualizations.graph_binary_objective_vs_threshold(pipeline, X, y, objective, steps=100)`

Generates a plot graphing objective score vs. decision thresholds for a fitted binary classification pipeline.

**Parameters**

- **pipeline** (*PipelineBase or subclass*) – Fitted pipeline
- **X** (*pd.DataFrame*) – The input data used to score and compute scores
- **y** (*pd.Series*) – The target labels
- **objective** (*ObjectiveBase obj, str*) – Objective used to score, shown on the y-axis of the graph
- **steps** (*int*) – Number of intervals to divide and calculate objective score at

**Returns** plotly.Figure representing the objective score vs. threshold graph generated

`evalml.model_understanding.visualizations.graph_prediction_vs_actual(y_true, y_pred, outlier_threshold=None)`

Generate a scatter plot comparing the true and predicted values. Used for regression plotting.

**Parameters**

- **y_true** (*pd.Series*) – The real target values of the data.
- **y_pred** (*pd.Series*) – The predicted values outputted by the regression model.

- **outlier_threshold** (*int, float*) – A positive threshold for what is considered an outlier value. This value is compared to the absolute difference between each value of y_true and y_pred. Values within this threshold will be blue, otherwise they will be yellow. Defaults to None.

**Returns** plotly.Figure representing the predicted vs. actual values graph

**Raises** **ValueError** – If threshold is not positive.

evalml.model_understanding.visualizations.**graph_prediction_vs_actual_over_time**(*pipeline*, *X*, *y*, *X_train*, *y_train*, *dates*)

Plot the target values and predictions against time on the x-axis.

**Parameters**

- **pipeline** (*TimeSeriesRegressionPipeline*) – Fitted time series regression pipeline.
- **X** (*pd.DataFrame*) – Features used to generate new predictions.
- **y** (*pd.Series*) – Target values to compare predictions against.
- **X_train** (*pd.DataFrame*) – Data the pipeline was trained on.
- **y_train** (*pd.Series*) – Target values for training data.
- **dates** (*pd.Series*) – Dates corresponding to target values and predictions.

**Returns** Showing the prediction vs actual over time.

**Return type** plotly.Figure

**Raises** **ValueError** – If the pipeline is not a time-series regression pipeline.

evalml.model_understanding.visualizations.**graph_t_sne**(*X*, *n_components=2*, *perplexity=30.0*, *learning_rate=200.0*, *metric='euclidean'*, *marker_line_width=2*, *marker_size=7*, ***kwargs*)

Plot high dimensional data into lower dimensional space using t-SNE.

**Parameters**

- **X** (*np.ndarray, pd.DataFrame*) – Data to be transformed. Must be numeric.
- **n_components** (*int*) – Dimension of the embedded space. Defaults to 2.
- **perplexity** (*float*) – Related to the number of nearest neighbors that is used in other manifold learning algorithms. Larger datasets usually require a larger perplexity. Consider selecting a value between 5 and 50. Defaults to 30.
- **learning_rate** (*float*) – Usually in the range [10.0, 1000.0]. If the cost function gets stuck in a bad local minimum, increasing the learning rate may help. Must be positive. Defaults to 200.
- **metric** (*str*) – The metric to use when calculating distance between instances in a feature array. The default is "euclidean" which is interpreted as the squared euclidean distance.
- **marker_line_width** (*int*) – Determines the line width of the marker boundary. Defaults to 2.
- **marker_size** (*int*) – Determines the size of the marker. Defaults to 7.
- **kwargs** – Arbitrary keyword arguments.

**Returns** Figure representing the transformed data.

**Return type** plotly.Figure

**Raises** `ValueError` – If marker_line_width or marker_size are not valid values.

`evalml.model_understanding.visualizations.t_sne(X, n_components=2, perplexity=30.0, learning_rate=200.0, metric='euclidean', **kwargs)`

Get the transformed output after fitting X to the embedded space using t-SNE.

**Parameters**

- **X** (*np.ndarray, pd.DataFrame*) – Data to be transformed. Must be numeric.
- **n_components** (*int, optional*) – Dimension of the embedded space.
- **perplexity** (*float, optional*) – Related to the number of nearest neighbors that is used in other manifold learning algorithms. Larger datasets usually require a larger perplexity. Consider selecting a value between 5 and 50.
- **learning_rate** (*float, optional*) – Usually in the range [10.0, 1000.0]. If the cost function gets stuck in a bad local minimum, increasing the learning rate may help.
- **metric** (*str, optional*) – The metric to use when calculating distance between instances in a feature array.
- **kwargs** – Arbitrary keyword arguments.

**Returns** TSNE output.

**Return type** np.ndarray (n_samples, n_components)

**Raises** `ValueError` – If specified parameters are not valid values.

`evalml.model_understanding.visualizations.visualize_decision_tree(estimator, max_depth=None, rotate=False, filled=False, filepath=None)`

Generate an image visualizing the decision tree.

**Parameters**

- **estimator** (*ComponentBase*) – A fitted DecisionTree-based estimator.
- **max_depth** (*int, optional*) – The depth to which the tree should be displayed. If set to None (as by default), tree is fully generated.
- **rotate** (*bool, optional*) – Orient tree left to right rather than top-down.
- **filled** (*bool, optional*) – Paint nodes to indicate majority class for classification, extremity of values for regression, or purity of node for multi-output.
- **filepath** (*str, optional*) – Path to where the graph should be saved. If set to None (as by default), the graph will not be saved.

**Returns** DOT object that can be directly displayed in Jupyter notebooks.

**Return type** graphviz.Source

**Raises**

- `ValueError` – If estimator is not a decision tree-based estimator.
- `NotFittedError` – If estimator is not yet fitted.

### Package Contents

### Functions

| | |
|---|---|
| binary_objective_vs_threshold | Computes objective score as a function of potential binary classification decision thresholds for a fitted binary classification pipeline. |
| calculate_permutation_importance | Calculates permutation importance for features. |
| calculate_permutation_importance_one_column | Calculates permutation importance for one column in the original dataframe. |
| confusion_matrix | Confusion matrix for binary and multiclass classification. |
| explain_predictions | Creates a report summarizing the top contributing features for each data point in the input features. |
| explain_predictions_best_worst | Creates a report summarizing the top contributing features for the best and worst points in the dataset as measured by error to true labels. |
| find_confusion_matrix_per_thresholds | Gets the confusion matrix and histogram bins for each threshold as well as the best threshold per objective. Only works with Binary Classification Pipelines. |
| get_linear_coefficients | Returns a dataframe showing the features with the greatest predictive power for a linear model. |
| get_prediction_vs_actual_data | Combines y_true and y_pred into a single dataframe and adds a column for outliers. Used in *graph_prediction_vs_actual()*. |
| get_prediction_vs_actual_over_time_data | Get the data needed for the prediction_vs_actual_over_time plot. |
| graph_binary_objective_vs_threshold | Generates a plot graphing objective score vs. decision thresholds for a fitted binary classification pipeline. |
| graph_confusion_matrix | Generate and display a confusion matrix plot. |
| graph_partial_dependence | Create an one-way or two-way partial dependence plot. |
| graph_permutation_importance | Generate a bar graph of the pipeline's permutation importance. |
| graph_precision_recall_curve | Generate and display a precision-recall plot. |
| graph_prediction_vs_actual | Generate a scatter plot comparing the true and predicted values. Used for regression plotting. |
| graph_prediction_vs_actual_over_time | Plot the target values and predictions against time on the x-axis. |
| graph_roc_curve | Generate and display a Receiver Operating Characteristic (ROC) plot for binary and multiclass classification problems. |
| graph_t_sne | Plot high dimensional data into lower dimensional space using t-SNE. |
| normalize_confusion_matrix | Normalizes a confusion matrix. |
| partial_dependence | Calculates one or two-way partial dependence. |
| precision_recall_curve | Given labels and binary classifier predicted probabilities, compute and return the data representing a precision-recall curve. |
| roc_curve | Given labels and classifier predicted probabilities, compute and return the data representing a Receiver Operating Characteristic (ROC) curve. Works with binary or multiclass problems. |
| t_sne | Get the transformed output after fitting X to the embedded space using t-SNE. |

## Contents

evalml.model_understanding.**binary_objective_vs_threshold**(*pipeline*, *X*, *y*, *objective*, *steps=100*)

Computes objective score as a function of potential binary classification decision thresholds for a fitted binary classification pipeline.

**Parameters**

- **pipeline** (*BinaryClassificationPipeline obj*) – Fitted binary classification pipeline.
- **X** (*pd.DataFrame*) – The input data used to compute objective score.
- **y** (*pd.Series*) – The target labels.
- **objective** (*ObjectiveBase obj, str*) – Objective used to score.
- **steps** (*int*) – Number of intervals to divide and calculate objective score at.

**Returns** DataFrame with thresholds and the corresponding objective score calculated at each threshold.

**Return type** pd.DataFrame

**Raises**

- **ValueError** – If objective is not a binary classification objective.
- **ValueError** – If objective's *score_needs_proba* is not False.

evalml.model_understanding.**calculate_permutation_importance**(*pipeline*, *X*, *y*, *objective*, *n_repeats=5*, *n_jobs=None*, *random_seed=0*)

Calculates permutation importance for features.

**Parameters**

- **pipeline** (*PipelineBase or subclass*) – Fitted pipeline.
- **X** (*pd.DataFrame*) – The input data used to score and compute permutation importance.
- **y** (*pd.Series*) – The target data.
- **objective** (*str, ObjectiveBase*) – Objective to score on.
- **n_repeats** (*int*) – Number of times to permute a feature. Defaults to 5.
- **n_jobs** (*int or None*) – Non-negative integer describing level of parallelism used for pipelines. None and 1 are equivalent. If set to -1, all CPUs are used. For n_jobs below -1, (n_cpus + 1 + n_jobs) are used. Defaults to None.
- **random_seed** (*int*) – Seed for the random number generator. Defaults to 0.

**Returns** Mean feature importance scores over a number of shuffles.

**Return type** pd.DataFrame

**Raises** **ValueError** – If objective cannot be used with the given pipeline.

evalml.model_understanding.**calculate_permutation_importance_one_column**(*pipeline*, *X*, *y*, *col_name*, *objective*, *n_repeats=5*, *fast=True*, *precomputed_features=None*, *random_seed=0*)

Calculates permutation importance for one column in the original dataframe.

**Parameters**

- **pipeline** (*PipelineBase or subclass*) – Fitted pipeline.
- **X** (*pd.DataFrame*) – The input data used to score and compute permutation importance.
- **y** (*pd.Series*) – The target data.
- **col_name** (*str, int*) – The column in X to calculate permutation importance for.
- **objective** (*str, ObjectiveBase*) – Objective to score on.
- **n_repeats** (*int*) – Number of times to permute a feature. Defaults to 5.
- **fast** (*bool*) – Whether to use the fast method of calculating the permutation importance or not. Defaults to True.
- **precomputed_features** (*pd.DataFrame*) – Precomputed features necessary to calculate permutation importance using the fast method. Defaults to None.
- **random_seed** (*int*) – Seed for the random number generator. Defaults to 0.

**Returns** Mean feature importance scores over a number of shuffles.

**Return type** float

**Raises**

- **ValueError** – If pipeline does not support fast permutation importance calculation.
- **ValueError** – If precomputed_features is None.

evalml.model_understanding.**confusion_matrix**(*y_true*, *y_predicted*, *normalize_method='true'*)

Confusion matrix for binary and multiclass classification.

**Parameters**

- **y_true** (*pd.Series or np.ndarray*) – True binary labels.
- **y_predicted** (*pd.Series or np.ndarray*) – Predictions from a binary classifier.
- **normalize_method** (*{'true', 'pred', 'all', None}*) – Normalization method to use, if not None. Supported options are: 'true' to normalize by row, 'pred' to normalize by column, or 'all' to normalize by all values. Defaults to 'true'.

**Returns** Confusion matrix. The column header represents the predicted labels while row header represents the actual labels.

**Return type** pd.DataFrame

evalml.model_understanding.**explain_predictions**(*pipeline*, *input_features*, *y*, *indices_to_explain*, *top_k_features=3*, *include_explainer_values=False*, *include_expected_value=False*, *output_format='text'*, *training_data=None*, *training_target=None*, *algorithm='shap'*)

Creates a report summarizing the top contributing features for each data point in the input features.

XGBoost models and CatBoost multiclass classifiers are not currently supported with the SHAP algorithm. To explain XGBoost model predictions, use the LIME algorithm. The LIME algorithm does not currently support any CatBoost models. For Stacked Ensemble models, the SHAP value for each input pipeline's predict function into the metalearner is used.

**Parameters**

- **pipeline** (*PipelineBase*) – Fitted pipeline whose predictions we want to explain with SHAP or LIME.

- **input_features** (*pd.DataFrame*) – Dataframe of input data to evaluate the pipeline on.
- **y** (*pd.Series*) – Labels for the input data.
- **indices_to_explain** (*list[int]*) – List of integer indices to explain.
- **top_k_features** (*int*) – How many of the highest/lowest contributing feature to include in the table for each data point. Default is 3.
- **include_explainer_values** (*bool*) – Whether explainer (SHAP or LIME) values should be included in the table. Default is False.
- **include_expected_value** (*bool*) – Whether the expected value should be included in the table. Default is False.
- **output_format** (*str*) – Either "text", "dict", or "dataframe". Default is "text".
- **training_data** (*pd.DataFrame, np.ndarray*) – Data the pipeline was trained on. Required and only used for time series pipelines.
- **training_target** (*pd.Series, np.ndarray*) – Targets used to train the pipeline. Required and only used for time series pipelines.
- **algorithm** (*str*) – Algorithm to use while generating top contributing features, one of "shap" or "lime". Defaults to "shap".

**Returns**

**A report explaining the top contributing features to each prediction for each row of input_features.** The report will include the feature names, prediction contribution, and explainer value (optional).

**Return type** str, dict, or pd.DataFrame

**Raises**

- **ValueError** – if input_features is empty.
- **ValueError** – if an output_format outside of "text", "dict" or "dataframe is provided.
- **ValueError** – if the requested index falls outside the input_feature's boundaries.

evalml.model_understanding.**explain_predictions_best_worst**(*pipeline*, *input_features*, *y_true*, *num_to_explain=5*, *top_k_features=3*, *include_explainer_values=False*, *metric=None*, *output_format='text'*, *callback=None*, *training_data=None*, *training_target=None*, *algorithm='shap'*)

Creates a report summarizing the top contributing features for the best and worst points in the dataset as measured by error to true labels.

XGBoost models and CatBoost multiclass classifiers are not currently supported with the SHAP algorithm. To explain XGBoost model predictions, use the LIME algorithm. The LIME algorithm does not currently support any CatBoost models. For Stacked Ensemble models, the SHAP value for each input pipeline's predict function into the metalearner is used.

**Parameters**

- **pipeline** (*PipelineBase*) – Fitted pipeline whose predictions we want to explain with SHAP or LIME.
- **input_features** (*pd.DataFrame*) – Input data to evaluate the pipeline on.
- **y_true** (*pd.Series*) – True labels for the input data.

- **num_to_explain** (*int*) – How many of the best, worst, random data points to explain.
- **top_k_features** (*int*) – How many of the highest/lowest contributing feature to include in the table for each data point.
- **include_explainer_values** (*bool*) – Whether explainer (SHAP or LIME) values should be included in the table. Default is False.
- **metric** (*callable*) – The metric used to identify the best and worst points in the dataset. Function must accept the true labels and predicted value or probabilities as the only arguments and lower values must be better. By default, this will be the absolute error for regression problems and cross entropy loss for classification problems.
- **output_format** (*str*) – Either "text" or "dict". Default is "text".
- **callback** (*callable*) – Function to be called with incremental updates. Has the following parameters: - progress_stage: stage of computation - time_elapsed: total time in seconds that has elapsed since start of call
- **training_data** (*pd.DataFrame, np.ndarray*) – Data the pipeline was trained on. Required and only used for time series pipelines.
- **training_target** (*pd.Series, np.ndarray*) – Targets used to train the pipeline. Required and only used for time series pipelines.
- **algorithm** (*str*) – Algorithm to use while generating top contributing features, one of "shap" or "lime". Defaults to "shap".

**Returns**

**A report explaining the top contributing features for the best/worst predictions in the input_features.** For each of the best/worst rows of input_features, the predicted values, true labels, metric value, feature names, prediction contribution, and explainer value (optional) will be listed.

**Return type** str, dict, or pd.DataFrame

**Raises**

- **ValueError** – If input_features does not have more than twice the requested features to explain.
- **ValueError** – If y_true and input_features have mismatched lengths.
- **ValueError** – If an output_format outside of "text", "dict" or "dataframe is provided.
- **PipelineScoreError** – If the pipeline errors out while scoring.

evalml.model_understanding.**find_confusion_matrix_per_thresholds**(*pipeline*, *X*, *y*, *n_bins=None*, *top_k=5*, *to_json=False*)

Gets the confusion matrix and histogram bins for each threshold as well as the best threshold per objective. Only works with Binary Classification Pipelines.

**Parameters**

- **pipeline** (*PipelineBase*) – A fitted Binary Classification Pipeline to get the confusion matrix with.
- **X** (*pd.DataFrame*) – The input features.
- **y** (*pd.Series*) – The input target.
- **n_bins** (*int*) – The number of bins to use to calculate the threshold values. Defaults to None, which will default to using Freedman-Diaconis rule.

- **top_k** (*int*) – The maximum number of row indices per bin to include as samples. -1 includes all row indices that fall between the bins. Defaults to 5.
- **to_json** (*bool*) – Whether or not to return a json output. If False, returns the (DataFrame, dict) tuple, otherwise returns a json.

**Returns**

**The dataframe has the actual positive histogram, actual negative histogram,** the confusion matrix, and a sample of rows that fall in the bin, all for each threshold value. The threshold value, represented through the dataframe index, represents the cutoff threshold at that value. The dictionary contains the ideal threshold and score per objective, keyed by objective name. If json, returns the info for both the dataframe and dictionary as a json output.

**Return type** (tuple(pd.DataFrame, dict)), json)

**Raises** **ValueError** – If the pipeline isn't a binary classification pipeline or isn't yet fitted on data.

evalml.model_understanding.**get_linear_coefficients**(*estimator*, *features=None*)

Returns a dataframe showing the features with the greatest predictive power for a linear model.

**Parameters**

- **estimator** (*Estimator*) – Fitted linear model family estimator.
- **features** (*list[str]*) – List of feature names associated with the underlying data.

**Returns** Displaying the features by importance.

**Return type** pd.DataFrame

**Raises**

- **ValueError** – If the model is not a linear model.
- **NotFittedError** – If the model is not yet fitted.

evalml.model_understanding.**get_prediction_vs_actual_data**(*y_true*, *y_pred*, *outlier_threshold=None*)

Combines y_true and y_pred into a single dataframe and adds a column for outliers. Used in *graph_prediction_vs_actual()*.

**Parameters**

- **y_true** (*pd.Series, or np.ndarray*) – The real target values of the data
- **y_pred** (*pd.Series, or np.ndarray*) – The predicted values outputted by the regression model.
- **outlier_threshold** (*int, float*) – A positive threshold for what is considered an outlier value. This value is compared to the absolute difference between each value of y_true and y_pred. Values within this threshold will be blue, otherwise they will be yellow. Defaults to None.

**Returns**

- *prediction*: Predicted values from regression model.
- *actual*: Real target values.
- *outlier*: Colors indicating which values are in the threshold for what is considered an outlier value.

**Return type** pd.DataFrame with the following columns

**Raises** **ValueError** – If threshold is not positive.

evalml.model_understanding.**get_prediction_vs_actual_over_time_data**(*pipeline*, *X*, *y*, *X_train*, *y_train*, *dates*)

Get the data needed for the prediction_vs_actual_over_time plot.

**Parameters**

- **pipeline** (*TimeSeriesRegressionPipeline*) – Fitted time series regression pipeline.
- **X** (*pd.DataFrame*) – Features used to generate new predictions.
- **y** (*pd.Series*) – Target values to compare predictions against.
- **X_train** (*pd.DataFrame*) – Data the pipeline was trained on.
- **y_train** (*pd.Series*) – Target values for training data.
- **dates** (*pd.Series*) – Dates corresponding to target values and predictions.

**Returns** Predictions vs. time.

**Return type** pd.DataFrame

evalml.model_understanding.**graph_binary_objective_vs_threshold**(*pipeline*, *X*, *y*, *objective*, *steps=100*)

Generates a plot graphing objective score vs. decision thresholds for a fitted binary classification pipeline.

**Parameters**

- **pipeline** (*PipelineBase or subclass*) – Fitted pipeline
- **X** (*pd.DataFrame*) – The input data used to score and compute scores
- **y** (*pd.Series*) – The target labels
- **objective** (*ObjectiveBase obj, str*) – Objective used to score, shown on the y-axis of the graph
- **steps** (*int*) – Number of intervals to divide and calculate objective score at

**Returns** plotly.Figure representing the objective score vs. threshold graph generated

evalml.model_understanding.**graph_confusion_matrix**(*y_true*, *y_pred*, *normalize_method='true'*, *title_addition=None*)

Generate and display a confusion matrix plot.

If *normalize_method* is set, hover text will show raw count, otherwise hover text will show count normalized with method 'true'.

**Parameters**

- **y_true** (*pd.Series or np.ndarray*) – True binary labels.
- **y_pred** (*pd.Series or np.ndarray*) – Predictions from a binary classifier.
- **normalize_method** (*{'true', 'pred', 'all', None}*) – Normalization method to use, if not None. Supported options are: 'true' to normalize by row, 'pred' to normalize by column, or 'all' to normalize by all values. Defaults to 'true'.
- **title_addition** (*str*) – If not None, append to plot title. Defaults to None.

**Returns** plotly.Figure representing the confusion matrix plot generated.

`evalml.model_understanding.graph_partial_dependence(pipeline, X, features, class_label=None, grid_resolution=100, kind='average')`

Create an one-way or two-way partial dependence plot.

Passing a single integer or string as features will create a one-way partial dependence plot with the feature values plotted against the partial dependence. Passing features a tuple of int/strings will create a two-way partial dependence plot with a contour of feature[0] in the y-axis, feature[1] in the x-axis and the partial dependence in the z-axis.

**Parameters**

- **pipeline** (*PipelineBase or subclass*) – Fitted pipeline.
- **X** (*pd.DataFrame, np.ndarray*) – The input data used to generate a grid of values for feature where partial dependence will be calculated at.
- **features** (*int, string, tuple[int or string]*) – The target feature for which to create the partial dependence plot for. If features is an int, it must be the index of the feature to use. If features is a string, it must be a valid column name in X. If features is a tuple of strings, it must contain valid column int/names in X.
- **class_label** (*string, optional*) – Name of class to plot for multiclass problems. If None, will plot the partial dependence for each class. This argument does not change behavior for regression or binary classification pipelines. For binary classification, the partial dependence for the positive label will always be displayed. Defaults to None.
- **grid_resolution** (*int*) – Number of samples of feature(s) for partial dependence plot.
- **kind** (*{'average', 'individual', 'both'}*) – Type of partial dependence to plot. 'average' creates a regular partial dependence (PD) graph, 'individual' creates an individual conditional expectation (ICE) plot, and 'both' creates a single-figure PD and ICE plot. ICE plots can only be shown for one-way partial dependence plots.

**Returns** figure object containing the partial dependence data for plotting

**Return type** plotly.graph_objects.Figure

**Raises**

- **PartialDependenceError** – if a graph is requested for a class name that isn't present in the pipeline.
- **PartialDependenceError** – if an ICE plot is requested for a two-way partial dependence.

`evalml.model_understanding.graph_permutation_importance(pipeline, X, y, objective, importance_threshold=0)`

Generate a bar graph of the pipeline's permutation importance.

**Parameters**

- **pipeline** (*PipelineBase or subclass*) – Fitted pipeline.
- **X** (*pd.DataFrame*) – The input data used to score and compute permutation importance.
- **y** (*pd.Series*) – The target data.
- **objective** (*str, ObjectiveBase*) – Objective to score on.
- **importance_threshold** (*float, optional*) – If provided, graph features with a permutation importance whose absolute value is larger than importance_threshold. Defaults to 0.

**Returns** plotly.Figure, a bar graph showing features and their respective permutation importance.

**Raises** **ValueError** – If importance_threshold is not greater than or equal to 0.

evalml.model_understanding.**graph_precision_recall_curve**(*y_true*, *y_pred_proba*, *title_addition=None*)

Generate and display a precision-recall plot.

**Parameters**

- **y_true** (*pd.Series or np.ndarray*) – True binary labels.
- **y_pred_proba** (*pd.Series or np.ndarray*) – Predictions from a binary classifier, before thresholding has been applied. Note this should be the predicted probability for the "true" label.
- **title_addition** (*str or None*) – If not None, append to plot title. Defaults to None.

**Returns** plotly.Figure representing the precision-recall plot generated

evalml.model_understanding.**graph_prediction_vs_actual**(*y_true*, *y_pred*, *outlier_threshold=None*)

Generate a scatter plot comparing the true and predicted values. Used for regression plotting.

**Parameters**

- **y_true** (*pd.Series*) – The real target values of the data.
- **y_pred** (*pd.Series*) – The predicted values outputted by the regression model.
- **outlier_threshold** (*int, float*) – A positive threshold for what is considered an outlier value. This value is compared to the absolute difference between each value of y_true and y_pred. Values within this threshold will be blue, otherwise they will be yellow. Defaults to None.

**Returns** plotly.Figure representing the predicted vs. actual values graph

**Raises** **ValueError** – If threshold is not positive.

evalml.model_understanding.**graph_prediction_vs_actual_over_time**(*pipeline*, *X*, *y*, *X_train*, *y_train*, *dates*)

Plot the target values and predictions against time on the x-axis.

**Parameters**

- **pipeline** (*TimeSeriesRegressionPipeline*) – Fitted time series regression pipeline.
- **X** (*pd.DataFrame*) – Features used to generate new predictions.
- **y** (*pd.Series*) – Target values to compare predictions against.
- **X_train** (*pd.DataFrame*) – Data the pipeline was trained on.
- **y_train** (*pd.Series*) – Target values for training data.
- **dates** (*pd.Series*) – Dates corresponding to target values and predictions.

**Returns** Showing the prediction vs actual over time.

**Return type** plotly.Figure

**Raises** **ValueError** – If the pipeline is not a time-series regression pipeline.

evalml.model_understanding.**graph_roc_curve**(*y_true*, *y_pred_proba*, *custom_class_names=None*, *title_addition=None*)

Generate and display a Receiver Operating Characteristic (ROC) plot for binary and multiclass classification problems.

**Parameters**

- **y_true** (*pd.Series or np.ndarray*) – True labels.

- **y_pred_proba** (*pd.Series or np.ndarray*) – Predictions from a classifier, before thresholding has been applied. Note this should a one dimensional array with the predicted probability for the "true" label in the binary case.
- **custom_class_names** (*list or None*) – If not None, custom labels for classes. Defaults to None.
- **title_addition** (*str or None*) – if not None, append to plot title. Defaults to None.

**Returns** plotly.Figure representing the ROC plot generated

**Raises** **ValueError** – If the number of custom class names does not match number of classes in the input data.

evalml.model_understanding.**graph_t_sne**(*X*, *n_components=2*, *perplexity=30.0*, *learning_rate=200.0*, *metric='euclidean'*, *marker_line_width=2*, *marker_size=7*, ***kwargs*)

Plot high dimensional data into lower dimensional space using t-SNE.

**Parameters**

- **X** (*np.ndarray, pd.DataFrame*) – Data to be transformed. Must be numeric.
- **n_components** (*int*) – Dimension of the embedded space. Defaults to 2.
- **perplexity** (*float*) – Related to the number of nearest neighbors that is used in other manifold learning algorithms. Larger datasets usually require a larger perplexity. Consider selecting a value between 5 and 50. Defaults to 30.
- **learning_rate** (*float*) – Usually in the range [10.0, 1000.0]. If the cost function gets stuck in a bad local minimum, increasing the learning rate may help. Must be positive. Defaults to 200.
- **metric** (*str*) – The metric to use when calculating distance between instances in a feature array. The default is "euclidean" which is interpreted as the squared euclidean distance.
- **marker_line_width** (*int*) – Determines the line width of the marker boundary. Defaults to 2.
- **marker_size** (*int*) – Determines the size of the marker. Defaults to 7.
- **kwargs** – Arbitrary keyword arguments.

**Returns** Figure representing the transformed data.

**Return type** plotly.Figure

**Raises** **ValueError** – If marker_line_width or marker_size are not valid values.

evalml.model_understanding.**normalize_confusion_matrix**(*conf_mat*, *normalize_method='true'*)

Normalizes a confusion matrix.

**Parameters**

- **conf_mat** (*pd.DataFrame or np.ndarray*) – Confusion matrix to normalize.
- **normalize_method** (*{'true', 'pred', 'all'}*) – Normalization method. Supported options are: 'true' to normalize by row, 'pred' to normalize by column, or 'all' to normalize by all values. Defaults to 'true'.

**Returns** normalized version of the input confusion matrix. The column header represents the predicted labels while row header represents the actual labels.

**Return type** pd.DataFrame

**Raises** `ValueError` – If configuration is invalid, or if the sum of a given axis is zero and normalization by axis is specified.

`evalml.model_understanding.partial_dependence(pipeline, X, features, percentiles=(0.05, 0.95), grid_resolution=100, kind='average', fast_mode=False, X_train=None, y_train=None)`

Calculates one or two-way partial dependence.

If a single integer or string is given for features, one-way partial dependence is calculated. If a tuple of two integers or strings is given, two-way partial dependence is calculated with the first feature in the y-axis and second feature in the x-axis.

**Parameters**

- **pipeline** (*PipelineBase or subclass*) – Fitted pipeline
- **X** (*pd.DataFrame, np.ndarray*) – The input data used to generate a grid of values for feature where partial dependence will be calculated at
- **features** (*int, string, tuple[int or string]*) – The target feature for which to create the partial dependence plot for. If features is an int, it must be the index of the feature to use. If features is a string, it must be a valid column name in X. If features is a tuple of int/strings, it must contain valid column integers/names in X.
- **percentiles** (*tuple[float]*) – The lower and upper percentile used to create the extreme values for the grid. Must be in [0, 1]. Defaults to (0.05, 0.95).
- **grid_resolution** (*int*) – Number of samples of feature(s) for partial dependence plot. If this value is less than the maximum number of categories present in categorical data within X, it will be set to the max number of categories + 1. Defaults to 100.
- **kind** (*{'average', 'individual', 'both'}*) – The type of predictions to return. 'individual' will return the predictions for all of the points in the grid for each sample in X. 'average' will return the predictions for all of the points in the grid but averaged over all of the samples in X.
- **fast_mode** (*bool, optional*) – Whether or not performance optimizations should be used for partial dependence calculations. Defaults to False. Note that user-specified components may not produce correct partial dependence results, so fast mode should only be used with EvalML-native components. Additionally, some components are not compatible with fast mode; in those cases, an error will be raised indicating that fast mode should not be used.
- **X_train** (*pd.DataFrame, np.ndarray*) – The data that was used to train the original pipeline. Will be used in fast mode to train the cloned pipelines. Defaults to None.
- **y_train** (*pd.Series, np.ndarray*) – The target data that was used to train the original pipeline. Will be used in fast mode to train the cloned pipelines. Defaults to None.

**Returns**

When *kind='average'*: DataFrame with averaged predictions for all points in the grid averaged over all samples of X and the values used to calculate those predictions.

When *kind='individual'*: DataFrame with individual predictions for all points in the grid for each sample of X and the values used to calculate those predictions. If a two-way partial dependence is calculated, then the result is a list of DataFrames with each DataFrame representing one sample's predictions.

When *kind='both'*: A tuple consisting of the averaged predictions (in a DataFrame) over all samples of X and the individual predictions (in a list of DataFrames) for each sample of X.

In the one-way case: The dataframe will contain two columns, "feature_values" (grid points at which the partial dependence was calculated) and "partial_dependence" (the partial dependence at that feature value). For classification problems, there will be a third column called "class_label" (the class label for which the partial dependence was calculated). For binary classification, the partial dependence is only calculated for the "positive" class.

In the two-way case: The data frame will contain grid_resolution number of columns and rows where the index and column headers are the sampled values of the first and second features, respectively, used to make the partial dependence contour. The values of the data frame contain the partial dependence data for each feature value pair.

**Return type** pd.DataFrame, list(pd.DataFrame), or tuple(pd.DataFrame, list(pd.DataFrame))

**Raises**

- **ValueError** – Error during call to scikit-learn's partial dependence method.
- **Exception** – All other errors during calculation.
- **PartialDependenceError** – if the user provides a tuple of not exactly two features.
- **PartialDependenceError** – if the provided pipeline isn't fitted.
- **PartialDependenceError** – if the provided pipeline is a Baseline pipeline.
- **PartialDependenceError** – if any of the features passed in are completely NaN
- **PartialDependenceError** – if any of the features are low-variance. Defined as having one value occurring more than the upper percentile passed by the user. By default 95%.

`evalml.model_understanding.`**`precision_recall_curve`**(*y_true*, *y_pred_proba*, *pos_label_idx=- 1*)

Given labels and binary classifier predicted probabilities, compute and return the data representing a precision-recall curve.

**Parameters**

- **y_true** (*pd.Series or np.ndarray*) – True binary labels.
- **y_pred_proba** (*pd.Series or np.ndarray*) – Predictions from a binary classifier, before thresholding has been applied. Note this should be the predicted probability for the "true" label.
- **pos_label_idx** (*int*) – the column index corresponding to the positive class. If predicted probabilities are two-dimensional, this will be used to access the probabilities for the positive class.

**Returns**

Dictionary containing metrics used to generate a precision-recall plot, with the following keys:

- *precision*: Precision values.
- *recall*: Recall values.
- *thresholds*: Threshold values used to produce the precision and recall.
- *auc_score*: The area under the ROC curve.

**Return type** list

**Raises** **NoPositiveLabelException** – If predicted probabilities do not contain a column at the specified label.

`evalml.model_understanding.roc_curve(y_true, y_pred_proba)`

Given labels and classifier predicted probabilities, compute and return the data representing a Receiver Operating Characteristic (ROC) curve. Works with binary or multiclass problems.

**Parameters**

- **y_true** (*pd.Series or np.ndarray*) – True labels.
- **y_pred_proba** (*pd.Series or pd.DataFrame or np.ndarray*) – Predictions from a classifier, before thresholding has been applied.

**Returns**

**A list of dictionaries (with one for each class) is returned. Binary classification problems return a list with one di**

**Each dictionary contains metrics used to generate an ROC plot with the following keys:**

- *fpr_rate*: False positive rate.
- *tpr_rate*: True positive rate.
- *threshold*: Threshold values used to produce each pair of true/false positive rates.
- *auc_score*: The area under the ROC curve.

**Return type** list(dict)

`evalml.model_understanding.t_sne(X, n_components=2, perplexity=30.0, learning_rate=200.0, metric='euclidean', **kwargs)`

Get the transformed output after fitting X to the embedded space using t-SNE.

**Parameters**

- **X** (*np.ndarray, pd.DataFrame*) – Data to be transformed. Must be numeric.
- **n_components** (*int, optional*) – Dimension of the embedded space.
- **perplexity** (*float, optional*) – Related to the number of nearest neighbors that is used in other manifold learning algorithms. Larger datasets usually require a larger perplexity. Consider selecting a value between 5 and 50.
- **learning_rate** (*float, optional*) – Usually in the range [10.0, 1000.0]. If the cost function gets stuck in a bad local minimum, increasing the learning rate may help.
- **metric** (*str, optional*) – The metric to use when calculating distance between instances in a feature array.
- **kwargs** – Arbitrary keyword arguments.

**Returns** TSNE output.

**Return type** np.ndarray (n_samples, n_components)

**Raises** **ValueError** – If specified parameters are not valid values.

## Objectives

EvalML standard and custom objectives.

### Submodules

### binary_classification_objective

Base class for all binary classification objectives.

### Module Contents

### Classes Summary

| | |
|---|---|
| *BinaryClassificationObjective* | Base class for all binary classification objectives. |

### Contents

**class** evalml.objectives.binary_classification_objective.**BinaryClassificationObjective**

Base class for all binary classification objectives.

**Attributes**

| | |
|---|---|
| **problem_types** | [ProblemTypes.BINARY, ProblemTypes.TIME_SERIES_BINARY] |

**Methods**

| | |
|---|---|
| *calculate_percent_difference* | Calculate the percent difference between scores. |
| *can_optimize_threshold* | Returns a boolean determining if we can optimize the binary classification objective threshold. |
| *decision_function* | Apply a learned threshold to predicted probabilities to get predicted classes. |
| *expected_range* | Returns the expected range of the objective, which is not necessarily the possible ranges. |
| *greater_is_better* | Returns a boolean determining if a greater score indicates better model performance. |
| *is_bounded_like_percentage* | Returns whether this objective is bounded between 0 and 1, inclusive. |
| *is_defined_for_problem_type* | Returns whether or not an objective is defined for a problem type. |
| *name* | Returns a name describing the objective. |
| *objective_function* | Computes the relative value of the provided predictions compared to the actual labels, according a specified metric. |
| *optimize_threshold* | Learn a binary classification threshold which optimizes the current objective. |
| *perfect_score* | Returns the score obtained by evaluating this objective on a perfect model. |
| *positive_only* | If True, this objective is only valid for positive data. Defaults to False. |
| *score* | Returns a numerical score indicating performance based on the differences between the predicted and actual values. |
| *score_needs_proba* | Returns a boolean determining if the score() method needs probability estimates. |
| *validate_inputs* | Validate inputs for scoring. |

**classmethod calculate_percent_difference**(*cls*, *score*, *baseline_score*)

Calculate the percent difference between scores.

**Parameters**

- **score** (*float*) – A score. Output of the score method of this objective.
- **baseline_score** (*float*) – A score. Output of the score method of this objective. In practice, this is the score achieved on this objective with a baseline estimator.

**Returns**

**The percent difference between the scores. Note that for objectives that can be interpreted** as percentages, this will be the difference between the reference score and score. For all other objectives, the difference will be normalized by the reference score.

**Return type** float

**property can_optimize_threshold**(*cls*)

Returns a boolean determining if we can optimize the binary classification objective threshold.

This will be false for any objective that works directly with predicted probabilities, like log loss and AUC. Otherwise, it will be true.

**Returns** Whether or not an objective can be optimized.

**Return type** bool

**decision_function**(*self*, *ypred_proba*, *threshold=0.5*, *X=None*)

Apply a learned threshold to predicted probabilities to get predicted classes.

**Parameters**

- **ypred_proba** (*pd.Series, np.ndarray*) – The classifier's predicted probabilities
- **threshold** (*float, optional*) – Threshold used to make a prediction. Defaults to 0.5.
- **X** (*pd.DataFrame, optional*) – Any extra columns that are needed from training data.

**Returns** predictions

**property expected_range**(*cls*)

Returns the expected range of the objective, which is not necessarily the possible ranges.

For example, our expected R2 range is from [-1, 1], although the actual range is (-inf, 1].

**property greater_is_better**(*cls*)

Returns a boolean determining if a greater score indicates better model performance.

**property is_bounded_like_percentage**(*cls*)

Returns whether this objective is bounded between 0 and 1, inclusive.

**classmethod is_defined_for_problem_type**(*cls*, *problem_type*)

Returns whether or not an objective is defined for a problem type.

**property name**(*cls*)

Returns a name describing the objective.

**abstract classmethod objective_function**(*cls*, *y_true*, *y_predicted*, *X=None*, *sample_weight=None*)

Computes the relative value of the provided predictions compared to the actual labels, according a specified metric.

**Parameters**

- **y_predicted** (*pd.Series*) – Predicted values of length [n_samples]
- **y_true** (*pd.Series*) – Actual class labels of length [n_samples]
- **X** (*pd.DataFrame or np.ndarray*) – Extra data of shape [n_samples, n_features] necessary to calculate score
- **sample_weight** (*pd.DataFrame or np.ndarray*) – Sample weights used in computing objective value result

**Returns** Numerical value used to calculate score

**optimize_threshold**(*self*, *ypred_proba*, *y_true*, *X=None*)

Learn a binary classification threshold which optimizes the current objective.

**Parameters**

- **ypred_proba** (*pd.Series*) – The classifier's predicted probabilities
- **y_true** (*pd.Series*) – The ground truth for the predictions.
- **X** (*pd.DataFrame, optional*) – Any extra columns that are needed from training data.

**Returns** Optimal threshold for this objective.

**Raises** **RuntimeError** – If objective cannot be optimized.

**property perfect_score**(*cls*)

Returns the score obtained by evaluating this objective on a perfect model.

**positive_only**(*cls*)

If True, this objective is only valid for positive data. Defaults to False.

**score**(*self*, *y_true*, *y_predicted*, *X=None*, *sample_weight=None*)

Returns a numerical score indicating performance based on the differences between the predicted and actual values.

**Parameters**

- **y_predicted** (*pd.Series*) – Predicted values of length [n_samples]
- **y_true** (*pd.Series*) – Actual class labels of length [n_samples]
- **X** (*pd.DataFrame or np.ndarray*) – Extra data of shape [n_samples, n_features] necessary to calculate score
- **sample_weight** (*pd.DataFrame or np.ndarray*) – Sample weights used in computing objective value result

**Returns** score

**property score_needs_proba**(*cls*)

Returns a boolean determining if the score() method needs probability estimates.

This should be true for objectives which work with predicted probabilities, like log loss or AUC, and false for objectives which compare predicted class labels to the actual labels, like F1 or correlation.

**validate_inputs**(*self*, *y_true*, *y_predicted*)

Validate inputs for scoring.

### cost_benefit_matrix

Cost-benefit matrix objective.

### Module Contents

### Classes Summary

| | |
|---|---|
| *CostBenefitMatrix* | Score using a cost-benefit matrix. Scores quantify the benefits of a given value, so greater numeric scores represents a better score. Costs and scores can be negative, indicating that a value is not beneficial. For example, in the case of monetary profit, a negative cost and/or score represents loss of cash flow. |

## Contents

**class** evalml.objectives.cost_benefit_matrix.**CostBenefitMatrix**(*true_positive*, *true_negative*, *false_positive*, *false_negative*)

Score using a cost-benefit matrix. Scores quantify the benefits of a given value, so greater numeric scores represents a better score. Costs and scores can be negative, indicating that a value is not beneficial. For example, in the case of monetary profit, a negative cost and/or score represents loss of cash flow.

**Parameters**

- **true_positive** (*float*) – Cost associated with true positive predictions.
- **true_negative** (*float*) – Cost associated with true negative predictions.
- **false_positive** (*float*) – Cost associated with false positive predictions.
- **false_negative** (*float*) – Cost associated with false negative predictions.

**Attributes**

| | |
|---|---|
| **expected_range** | None |
| **greater_is_better** | True |
| **is_bounded_like_percentage** | False |
| **name** | Cost Benefit Matrix |
| **perfect_score** | None |
| **problem_types** | [ProblemTypes.BINARY, ProblemTypes.TIME_SERIES_BINARY] |
| **score_needs_proba** | False |

**Methods**

| | |
|---|---|
| *calculate_percent_difference* | Calculate the percent difference between scores. |
| *can_optimize_threshold* | Returns a boolean determining if we can optimize the binary classification objective threshold. |
| *decision_function* | Apply a learned threshold to predicted probabilities to get predicted classes. |
| *is_defined_for_problem_type* | Returns whether or not an objective is defined for a problem type. |
| *objective_function* | Calculates cost-benefit of the using the predicted and true values. |
| *optimize_threshold* | Learn a binary classification threshold which optimizes the current objective. |
| *positive_only* | If True, this objective is only valid for positive data. Defaults to False. |
| *score* | Returns a numerical score indicating performance based on the differences between the predicted and actual values. |
| *validate_inputs* | Validate inputs for scoring. |

**classmethod calculate_percent_difference**(*cls*, *score*, *baseline_score*)

Calculate the percent difference between scores.

**Parameters**

- **score** (*float*) – A score. Output of the score method of this objective.
- **baseline_score** (*float*) – A score. Output of the score method of this objective. In practice, this is the score achieved on this objective with a baseline estimator.

**Returns**

**The percent difference between the scores. Note that for objectives that can be interpreted** as percentages, this will be the difference between the reference score and score. For all other objectives, the difference will be normalized by the reference score.

**Return type** float

**property can_optimize_threshold**(*cls*)

Returns a boolean determining if we can optimize the binary classification objective threshold.

This will be false for any objective that works directly with predicted probabilities, like log loss and AUC. Otherwise, it will be true.

**Returns** Whether or not an objective can be optimized.

**Return type** bool

**decision_function**(*self*, *ypred_proba*, *threshold=0.5*, *X=None*)

Apply a learned threshold to predicted probabilities to get predicted classes.

**Parameters**

- **ypred_proba** (*pd.Series, np.ndarray*) – The classifier's predicted probabilities
- **threshold** (*float, optional*) – Threshold used to make a prediction. Defaults to 0.5.
- **X** (*pd.DataFrame, optional*) – Any extra columns that are needed from training data.

**Returns** predictions

**classmethod is_defined_for_problem_type**(*cls*, *problem_type*)

Returns whether or not an objective is defined for a problem type.

**objective_function**(*self*, *y_true*, *y_predicted*, *X=None*, *sample_weight=None*)

Calculates cost-benefit of the using the predicted and true values.

**Parameters**

- **y_predicted** (*pd.Series*) – Predicted labels.
- **y_true** (*pd.Series*) – True labels.
- **X** (*pd.DataFrame*) – Ignored.
- **sample_weight** (*pd.DataFrame*) – Ignored.

**Returns** Cost-benefit matrix score

**Return type** float

**optimize_threshold**(*self*, *ypred_proba*, *y_true*, *X=None*)

Learn a binary classification threshold which optimizes the current objective.

**Parameters**

- **ypred_proba** (*pd.Series*) – The classifier's predicted probabilities
- **y_true** (*pd.Series*) – The ground truth for the predictions.
- **X** (*pd.DataFrame, optional*) – Any extra columns that are needed from training data.

**Returns** Optimal threshold for this objective.

**Raises** `RuntimeError` – If objective cannot be optimized.

`positive_only`(*cls*)

If True, this objective is only valid for positive data. Defaults to False.

`score`(*self*, *y_true*, *y_predicted*, *X=None*, *sample_weight=None*)

Returns a numerical score indicating performance based on the differences between the predicted and actual values.

**Parameters**

- `y_predicted` (*pd.Series*) – Predicted values of length [n_samples]
- `y_true` (*pd.Series*) – Actual class labels of length [n_samples]
- `X` (*pd.DataFrame or np.ndarray*) – Extra data of shape [n_samples, n_features] necessary to calculate score
- `sample_weight` (*pd.DataFrame or np.ndarray*) – Sample weights used in computing objective value result

**Returns** score

`validate_inputs`(*self*, *y_true*, *y_predicted*)

Validate inputs for scoring.

## fraud_cost

Score the percentage of money lost of the total transaction amount process due to fraud.

### Module Contents

### Classes Summary

| | |
|---|---|
| *FraudCost* | Score the percentage of money lost of the total transaction amount process due to fraud. |

### Contents

**class** `evalml.objectives.fraud_cost.`**`FraudCost`**(*retry_percentage=0.5*, *interchange_fee=0.02*, *fraud_payout_percentage=1.0*, *amount_col='amount'*)

Score the percentage of money lost of the total transaction amount process due to fraud.

**Parameters**

- `retry_percentage` (*float*) – What percentage of customers that will retry a transaction if it is declined. Between 0 and 1. Defaults to 0.5.
- `interchange_fee` (*float*) – How much of each successful transaction you pay. Between 0 and 1. Defaults to 0.02.
- `fraud_payout_percentage` (*float*) – Percentage of fraud you will not be able to collect. Between 0 and 1. Defaults to 1.0.

- **amount_col** (*str*) – Name of column in data that contains the amount. Defaults to "amount".

**Attributes**

| | |
|---|---|
| **expected_range** | None |
| **greater_is_better** | False |
| **is_bounded_like_percentage** | True |
| **name** | Fraud Cost |
| **perfect_score** | 0.0 |
| **problem_types** | [ProblemTypes.BINARY, ProblemTypes.TIME_SERIES_BINARY] |
| **score_needs_proba** | False |

**Methods**

| | |
|---|---|
| *calculate_percent_difference* | Calculate the percent difference between scores. |
| *can_optimize_threshold* | Returns a boolean determining if we can optimize the binary classification objective threshold. |
| *decision_function* | Apply a learned threshold to predicted probabilities to get predicted classes. |
| *is_defined_for_problem_type* | Returns whether or not an objective is defined for a problem type. |
| *objective_function* | Calculate amount lost to fraud per transaction given predictions, true values, and dataframe with transaction amount. |
| *optimize_threshold* | Learn a binary classification threshold which optimizes the current objective. |
| *positive_only* | If True, this objective is only valid for positive data. Defaults to False. |
| *score* | Returns a numerical score indicating performance based on the differences between the predicted and actual values. |
| *validate_inputs* | Validate inputs for scoring. |

**classmethod calculate_percent_difference**(*cls*, *score*, *baseline_score*)

Calculate the percent difference between scores.

**Parameters**

- **score** (*float*) – A score. Output of the score method of this objective.
- **baseline_score** (*float*) – A score. Output of the score method of this objective. In practice, this is the score achieved on this objective with a baseline estimator.

**Returns**

**The percent difference between the scores. Note that for objectives that can be interpreted** as percentages, this will be the difference between the reference score and score. For all other objectives, the difference will be normalized by the reference score.

**Return type** float

**property can_optimize_threshold**(*cls*)

Returns a boolean determining if we can optimize the binary classification objective threshold.

This will be false for any objective that works directly with predicted probabilities, like log loss and AUC. Otherwise, it will be true.

**Returns** Whether or not an objective can be optimized.

**Return type** bool

**decision_function**(*self*, *ypred_proba*, *threshold=0.5*, *X=None*)

Apply a learned threshold to predicted probabilities to get predicted classes.

**Parameters**

- **ypred_proba** (*pd.Series, np.ndarray*) – The classifier's predicted probabilities
- **threshold** (*float, optional*) – Threshold used to make a prediction. Defaults to 0.5.
- **X** (*pd.DataFrame, optional*) – Any extra columns that are needed from training data.

**Returns** predictions

**classmethod is_defined_for_problem_type**(*cls*, *problem_type*)

Returns whether or not an objective is defined for a problem type.

**objective_function**(*self*, *y_true*, *y_predicted*, *X*, *sample_weight=None*)

Calculate amount lost to fraud per transaction given predictions, true values, and dataframe with transaction amount.

**Parameters**

- **y_predicted** (*pd.Series*) – Predicted fraud labels.
- **y_true** (*pd.Series*) – True fraud labels.
- **X** (*pd.DataFrame*) – Data with transaction amounts.
- **sample_weight** (*pd.DataFrame*) – Ignored.

**Returns** Amount lost to fraud per transaction.

**Return type** float

**Raises** **ValueError** – If amount_col is not a valid column in the input data.

**optimize_threshold**(*self*, *ypred_proba*, *y_true*, *X=None*)

Learn a binary classification threshold which optimizes the current objective.

**Parameters**

- **ypred_proba** (*pd.Series*) – The classifier's predicted probabilities
- **y_true** (*pd.Series*) – The ground truth for the predictions.
- **X** (*pd.DataFrame, optional*) – Any extra columns that are needed from training data.

**Returns** Optimal threshold for this objective.

**Raises** **RuntimeError** – If objective cannot be optimized.

**positive_only**(*cls*)

If True, this objective is only valid for positive data. Defaults to False.

**score**(*self*, *y_true*, *y_predicted*, *X=None*, *sample_weight=None*)

Returns a numerical score indicating performance based on the differences between the predicted and actual values.

**Parameters**

- **y_predicted** (*pd.Series*) – Predicted values of length [n_samples]
- **y_true** (*pd.Series*) – Actual class labels of length [n_samples]
- **X** (*pd.DataFrame or np.ndarray*) – Extra data of shape [n_samples, n_features] necessary to calculate score
- **sample_weight** (*pd.DataFrame or np.ndarray*) – Sample weights used in computing objective value result

**Returns** score

**validate_inputs**(*self*, *y_true*, *y_predicted*)

Validate inputs for scoring.

## lead_scoring

Lead scoring objective.

## Module Contents

## Classes Summary

| | |
|---|---|
| *LeadScoring* | Lead scoring. |

## Contents

**class** evalml.objectives.lead_scoring.**LeadScoring**(*true_positives=1*, *false_positives=- 1*)

Lead scoring.

**Parameters**

- **true_positives** (*int*) – Reward for a true positive. Defaults to 1.
- **false_positives** (*int*) – Cost for a false positive. Should be negative. Defaults to -1.

**Attributes**

| | |
|---|---|
| **expected_range** | None |
| **greater_is_better** | True |
| **is_bounded_like_percentage** | False |
| **name** | Lead Scoring |
| **perfect_score** | None |
| **problem_types** | [ProblemTypes.BINARY, ProblemTypes.TIME_SERIES_BINARY] |
| **score_needs_proba** | False |

**Methods**

| | |
|---|---|
| *calculate_percent_difference* | Calculate the percent difference between scores. |
| *can_optimize_threshold* | Returns a boolean determining if we can optimize the binary classification objective threshold. |
| *decision_function* | Apply a learned threshold to predicted probabilities to get predicted classes. |
| *is_defined_for_problem_type* | Returns whether or not an objective is defined for a problem type. |
| *objective_function* | Calculate the profit per lead. |
| *optimize_threshold* | Learn a binary classification threshold which optimizes the current objective. |
| *positive_only* | If True, this objective is only valid for positive data. Defaults to False. |
| *score* | Returns a numerical score indicating performance based on the differences between the predicted and actual values. |
| *validate_inputs* | Validate inputs for scoring. |

**classmethod calculate_percent_difference**(*cls*, *score*, *baseline_score*)

Calculate the percent difference between scores.

**Parameters**

- **score** (*float*) – A score. Output of the score method of this objective.
- **baseline_score** (*float*) – A score. Output of the score method of this objective. In practice, this is the score achieved on this objective with a baseline estimator.

**Returns**

**The percent difference between the scores. Note that for objectives that can be interpreted** as percentages, this will be the difference between the reference score and score. For all other objectives, the difference will be normalized by the reference score.

**Return type** float

**property can_optimize_threshold**(*cls*)

Returns a boolean determining if we can optimize the binary classification objective threshold.

This will be false for any objective that works directly with predicted probabilities, like log loss and AUC. Otherwise, it will be true.

**Returns** Whether or not an objective can be optimized.

**Return type** bool

**decision_function**(*self*, *ypred_proba*, *threshold=0.5*, *X=None*)

Apply a learned threshold to predicted probabilities to get predicted classes.

**Parameters**

- **ypred_proba** (*pd.Series, np.ndarray*) – The classifier's predicted probabilities
- **threshold** (*float, optional*) – Threshold used to make a prediction. Defaults to 0.5.
- **X** (*pd.DataFrame, optional*) – Any extra columns that are needed from training data.

**Returns** predictions

**classmethod is_defined_for_problem_type**(*cls*, *problem_type*)

Returns whether or not an objective is defined for a problem type.

**objective_function**(*self*, *y_true*, *y_predicted*, *X=None*, *sample_weight=None*)

Calculate the profit per lead.

**Parameters**

- **y_predicted** (*pd.Series*) – Predicted labels
- **y_true** (*pd.Series*) – True labels
- **X** (*pd.DataFrame*) – Ignored.
- **sample_weight** (*pd.DataFrame*) – Ignored.

**Returns** Profit per lead

**Return type** float

**optimize_threshold**(*self*, *ypred_proba*, *y_true*, *X=None*)

Learn a binary classification threshold which optimizes the current objective.

**Parameters**

- **ypred_proba** (*pd.Series*) – The classifier's predicted probabilities
- **y_true** (*pd.Series*) – The ground truth for the predictions.
- **X** (*pd.DataFrame, optional*) – Any extra columns that are needed from training data.

**Returns** Optimal threshold for this objective.

**Raises** **RuntimeError** – If objective cannot be optimized.

**positive_only**(*cls*)

If True, this objective is only valid for positive data. Defaults to False.

**score**(*self*, *y_true*, *y_predicted*, *X=None*, *sample_weight=None*)

Returns a numerical score indicating performance based on the differences between the predicted and actual values.

**Parameters**

- **y_predicted** (*pd.Series*) – Predicted values of length [n_samples]
- **y_true** (*pd.Series*) – Actual class labels of length [n_samples]
- **X** (*pd.DataFrame or np.ndarray*) – Extra data of shape [n_samples, n_features] necessary to calculate score
- **sample_weight** (*pd.DataFrame or np.ndarray*) – Sample weights used in computing objective value result

**Returns** score

**validate_inputs**(*self*, *y_true*, *y_predicted*)

Validate inputs for scoring.

### multiclass_classification_objective

Base class for all multiclass classification objectives.

### Module Contents

### Classes Summary

| | |
|---|---|
| *MulticlassClassificationObjective* | Base class for all multiclass classification objectives. |

### Contents

**class** evalml.objectives.multiclass_classification_objective.**MulticlassClassificationObjective**

Base class for all multiclass classification objectives.

**Attributes**

| | |
|---|---|
| **problem_types** | [ProblemTypes.MULTICLASS, ProblemTypes.TIME_SERIES_MULTICLASS] |

**Methods**

| | |
|---|---|
| *calculate_percent_difference* | Calculate the percent difference between scores. |
| *expected_range* | Returns the expected range of the objective, which is not necessarily the possible ranges. |
| *greater_is_better* | Returns a boolean determining if a greater score indicates better model performance. |
| *is_bounded_like_percentage* | Returns whether this objective is bounded between 0 and 1, inclusive. |
| *is_defined_for_problem_type* | Returns whether or not an objective is defined for a problem type. |
| *name* | Returns a name describing the objective. |
| *objective_function* | Computes the relative value of the provided predictions compared to the actual labels, according a specified metric. |
| *perfect_score* | Returns the score obtained by evaluating this objective on a perfect model. |
| *positive_only* | If True, this objective is only valid for positive data. Defaults to False. |
| *score* | Returns a numerical score indicating performance based on the differences between the predicted and actual values. |
| *score_needs_proba* | Returns a boolean determining if the score() method needs probability estimates. |
| *validate_inputs* | Validates the input based on a few simple checks. |

**classmethod calculate_percent_difference**(*cls*, *score*, *baseline_score*)

Calculate the percent difference between scores.

**Parameters**

- **score** (*float*) – A score. Output of the score method of this objective.
- **baseline_score** (*float*) – A score. Output of the score method of this objective. In practice, this is the score achieved on this objective with a baseline estimator.

**Returns**

**The percent difference between the scores. Note that for objectives that can be interpreted** as percentages, this will be the difference between the reference score and score. For all other objectives, the difference will be normalized by the reference score.

**Return type** float

**property expected_range**(*cls*)

Returns the expected range of the objective, which is not necessarily the possible ranges.

For example, our expected R2 range is from [-1, 1], although the actual range is (-inf, 1].

**property greater_is_better**(*cls*)

Returns a boolean determining if a greater score indicates better model performance.

**property is_bounded_like_percentage**(*cls*)

Returns whether this objective is bounded between 0 and 1, inclusive.

**classmethod is_defined_for_problem_type**(*cls*, *problem_type*)

Returns whether or not an objective is defined for a problem type.

**property name**(*cls*)

Returns a name describing the objective.

**abstract classmethod objective_function**(*cls*, *y_true*, *y_predicted*, *X=None*, *sample_weight=None*)

Computes the relative value of the provided predictions compared to the actual labels, according a specified metric.

**Parameters**

- **y_predicted** (*pd.Series*) – Predicted values of length [n_samples]
- **y_true** (*pd.Series*) – Actual class labels of length [n_samples]
- **X** (*pd.DataFrame or np.ndarray*) – Extra data of shape [n_samples, n_features] necessary to calculate score
- **sample_weight** (*pd.DataFrame or np.ndarray*) – Sample weights used in computing objective value result

**Returns** Numerical value used to calculate score

**property perfect_score**(*cls*)

Returns the score obtained by evaluating this objective on a perfect model.

**positive_only**(*cls*)

If True, this objective is only valid for positive data. Defaults to False.

**score**(*self*, *y_true*, *y_predicted*, *X=None*, *sample_weight=None*)

Returns a numerical score indicating performance based on the differences between the predicted and actual values.

**Parameters**

- **y_predicted** (*pd.Series*) – Predicted values of length [n_samples]

- **y_true** (*pd.Series*) – Actual class labels of length [n_samples]
- **X** (*pd.DataFrame or np.ndarray*) – Extra data of shape [n_samples, n_features] necessary to calculate score
- **sample_weight** (*pd.DataFrame or np.ndarray*) – Sample weights used in computing objective value result

**Returns** score

**property score_needs_proba**(*cls*)

Returns a boolean determining if the score() method needs probability estimates.

This should be true for objectives which work with predicted probabilities, like log loss or AUC, and false for objectives which compare predicted class labels to the actual labels, like F1 or correlation.

**validate_inputs**(*self*, *y_true*, *y_predicted*)

Validates the input based on a few simple checks.

**Parameters**

- **y_predicted** (*pd.Series, or pd.DataFrame*) – Predicted values of length [n_samples].
- **y_true** (*pd.Series*) – Actual class labels of length [n_samples].

**Raises** **ValueError** – If the inputs are malformed.

## objective_base

Base class for all objectives.

## Module Contents

## Classes Summary

| | |
|---|---|
| *ObjectiveBase* | Base class for all objectives. |

## Contents

**class** evalml.objectives.objective_base.**ObjectiveBase**

Base class for all objectives.

**Attributes**

| | |
|---|---|
| **problem_types** | None |

**Methods**

| | |
|---|---|
| *calculate_percent_difference* | Calculate the percent difference between scores. |
| *expected_range* | Returns the expected range of the objective, which is not necessarily the possible ranges. |
| *greater_is_better* | Returns a boolean determining if a greater score indicates better model performance. |
| *is_bounded_like_percentage* | Returns whether this objective is bounded between 0 and 1, inclusive. |
| *is_defined_for_problem_type* | Returns whether or not an objective is defined for a problem type. |
| *name* | Returns a name describing the objective. |
| *objective_function* | Computes the relative value of the provided predictions compared to the actual labels, according a specified metric. |
| *perfect_score* | Returns the score obtained by evaluating this objective on a perfect model. |
| *positive_only* | If True, this objective is only valid for positive data. Defaults to False. |
| *score* | Returns a numerical score indicating performance based on the differences between the predicted and actual values. |
| *score_needs_proba* | Returns a boolean determining if the score() method needs probability estimates. |
| *validate_inputs* | Validates the input based on a few simple checks. |

**classmethod calculate_percent_difference**(*cls*, *score*, *baseline_score*)

Calculate the percent difference between scores.

**Parameters**

- **score** (*float*) – A score. Output of the score method of this objective.
- **baseline_score** (*float*) – A score. Output of the score method of this objective. In practice, this is the score achieved on this objective with a baseline estimator.

**Returns**

**The percent difference between the scores. Note that for objectives that can be interpreted** as percentages, this will be the difference between the reference score and score. For all other objectives, the difference will be normalized by the reference score.

**Return type** float

**property expected_range**(*cls*)

Returns the expected range of the objective, which is not necessarily the possible ranges.

For example, our expected R2 range is from [-1, 1], although the actual range is (-inf, 1].

**property greater_is_better**(*cls*)

Returns a boolean determining if a greater score indicates better model performance.

**property is_bounded_like_percentage**(*cls*)

Returns whether this objective is bounded between 0 and 1, inclusive.

**classmethod is_defined_for_problem_type**(*cls*, *problem_type*)

Returns whether or not an objective is defined for a problem type.

**property name**(*cls*)

Returns a name describing the objective.

**abstract classmethod objective_function**(*cls*, *y_true*, *y_predicted*, *X=None*, *sample_weight=None*)

Computes the relative value of the provided predictions compared to the actual labels, according a specified metric.

**Parameters**

- **y_predicted** (*pd.Series*) – Predicted values of length [n_samples]
- **y_true** (*pd.Series*) – Actual class labels of length [n_samples]
- **X** (*pd.DataFrame or np.ndarray*) – Extra data of shape [n_samples, n_features] necessary to calculate score
- **sample_weight** (*pd.DataFrame or np.ndarray*) – Sample weights used in computing objective value result

**Returns** Numerical value used to calculate score

**property perfect_score**(*cls*)

Returns the score obtained by evaluating this objective on a perfect model.

**positive_only**(*cls*)

If True, this objective is only valid for positive data. Defaults to False.

**score**(*self*, *y_true*, *y_predicted*, *X=None*, *sample_weight=None*)

Returns a numerical score indicating performance based on the differences between the predicted and actual values.

**Parameters**

- **y_predicted** (*pd.Series*) – Predicted values of length [n_samples]
- **y_true** (*pd.Series*) – Actual class labels of length [n_samples]
- **X** (*pd.DataFrame or np.ndarray*) – Extra data of shape [n_samples, n_features] necessary to calculate score
- **sample_weight** (*pd.DataFrame or np.ndarray*) – Sample weights used in computing objective value result

**Returns** score

**property score_needs_proba**(*cls*)

Returns a boolean determining if the score() method needs probability estimates.

This should be true for objectives which work with predicted probabilities, like log loss or AUC, and false for objectives which compare predicted class labels to the actual labels, like F1 or correlation.

**validate_inputs**(*self*, *y_true*, *y_predicted*)

Validates the input based on a few simple checks.

**Parameters**

- **y_predicted** (*pd.Series, or pd.DataFrame*) – Predicted values of length [n_samples].
- **y_true** (*pd.Series*) – Actual class labels of length [n_samples].

**Raises** **ValueError** – If the inputs are malformed.

### regression_objective

Base class for all regression objectives.

### Module Contents

### Classes Summary

| | |
|---|---|
| *RegressionObjective* | Base class for all regression objectives. |

### Contents

**class** evalml.objectives.regression_objective.**RegressionObjective**

Base class for all regression objectives.

**Attributes**

| | |
|---|---|
| **problem_types** | [ProblemTypes.REGRESSION, ProblemTypes.TIME_SERIES_REGRESSION] |

**Methods**

| | |
|---|---|
| *calculate_percent_difference* | Calculate the percent difference between scores. |
| *expected_range* | Returns the expected range of the objective, which is not necessarily the possible ranges. |
| *greater_is_better* | Returns a boolean determining if a greater score indicates better model performance. |
| *is_bounded_like_percentage* | Returns whether this objective is bounded between 0 and 1, inclusive. |
| *is_defined_for_problem_type* | Returns whether or not an objective is defined for a problem type. |
| *name* | Returns a name describing the objective. |
| *objective_function* | Computes the relative value of the provided predictions compared to the actual labels, according a specified metric. |
| *perfect_score* | Returns the score obtained by evaluating this objective on a perfect model. |
| *positive_only* | If True, this objective is only valid for positive data. Defaults to False. |
| *score* | Returns a numerical score indicating performance based on the differences between the predicted and actual values. |
| *score_needs_proba* | Returns a boolean determining if the score() method needs probability estimates. |
| *validate_inputs* | Validates the input based on a few simple checks. |

**classmethod calculate_percent_difference**(*cls*, *score*, *baseline_score*)

Calculate the percent difference between scores.

**Parameters**

- **score** (*float*) – A score. Output of the score method of this objective.
- **baseline_score** (*float*) – A score. Output of the score method of this objective. In practice, this is the score achieved on this objective with a baseline estimator.

**Returns**

**The percent difference between the scores. Note that for objectives that can be interpreted** as percentages, this will be the difference between the reference score and score. For all other objectives, the difference will be normalized by the reference score.

**Return type** float

**property expected_range**(*cls*)

Returns the expected range of the objective, which is not necessarily the possible ranges.

For example, our expected R2 range is from [-1, 1], although the actual range is (-inf, 1].

**property greater_is_better**(*cls*)

Returns a boolean determining if a greater score indicates better model performance.

**property is_bounded_like_percentage**(*cls*)

Returns whether this objective is bounded between 0 and 1, inclusive.

**classmethod is_defined_for_problem_type**(*cls*, *problem_type*)

Returns whether or not an objective is defined for a problem type.

**property name**(*cls*)

Returns a name describing the objective.

**abstract classmethod objective_function**(*cls*, *y_true*, *y_predicted*, *X=None*, *sample_weight=None*)

Computes the relative value of the provided predictions compared to the actual labels, according a specified metric.

**Parameters**

- **y_predicted** (*pd.Series*) – Predicted values of length [n_samples]
- **y_true** (*pd.Series*) – Actual class labels of length [n_samples]
- **X** (*pd.DataFrame or np.ndarray*) – Extra data of shape [n_samples, n_features] necessary to calculate score
- **sample_weight** (*pd.DataFrame or np.ndarray*) – Sample weights used in computing objective value result

**Returns** Numerical value used to calculate score

**property perfect_score**(*cls*)

Returns the score obtained by evaluating this objective on a perfect model.

**positive_only**(*cls*)

If True, this objective is only valid for positive data. Defaults to False.

**score**(*self*, *y_true*, *y_predicted*, *X=None*, *sample_weight=None*)

Returns a numerical score indicating performance based on the differences between the predicted and actual values.

**Parameters**

- **y_predicted** (*pd.Series*) – Predicted values of length [n_samples]
- **y_true** (*pd.Series*) – Actual class labels of length [n_samples]

- **X** (*pd.DataFrame or np.ndarray*) – Extra data of shape [n_samples, n_features] necessary to calculate score
- **sample_weight** (*pd.DataFrame or np.ndarray*) – Sample weights used in computing objective value result

**Returns** score

**property score_needs_proba**(*cls*)

Returns a boolean determining if the score() method needs probability estimates.

This should be true for objectives which work with predicted probabilities, like log loss or AUC, and false for objectives which compare predicted class labels to the actual labels, like F1 or correlation.

**validate_inputs**(*self*, *y_true*, *y_predicted*)

Validates the input based on a few simple checks.

**Parameters**

- **y_predicted** (*pd.Series, or pd.DataFrame*) – Predicted values of length [n_samples].
- **y_true** (*pd.Series*) – Actual class labels of length [n_samples].

**Raises** **ValueError** – If the inputs are malformed.

## sensitivity_low_alert

Sensitivity at Low Alert Rates objective.

### Module Contents

### Classes Summary

| | |
|---|---|
| *SensitivityLowAlert* | Create instance of SensitivityLowAlert. |

### Attributes Summary

| |
|---|
| *logger* |

### Contents

evalml.objectives.sensitivity_low_alert.**logger**

**class** evalml.objectives.sensitivity_low_alert.**SensitivityLowAlert**(*alert_rate=0.01*)

Create instance of SensitivityLowAlert.

**Parameters** **alert_rate** (*float*) – percentage of top scores to classify as high risk.

**Attributes**

| | |
|---|---|
| **expected_range** | [0, 1] |
| **greater_is_better** | True |
| **is_bounded_like_percentage** | True |
| **name** | Sensitivity at Low Alert Rates |
| **perfect_score** | 1.0 |
| **problem_types** | [ProblemTypes.BINARY, ProblemTypes.TIME_SERIES_BINARY] |
| **score_needs_proba** | False |

**Methods**

| | |
|---|---|
| *calculate_percent_difference* | Calculate the percent difference between scores. |
| *can_optimize_threshold* | Returns a boolean determining if we can optimize the binary classification objective threshold. |
| *decision_function* | Determine if an observation is high risk given an alert rate. |
| *is_defined_for_problem_type* | Returns whether or not an objective is defined for a problem type. |
| *objective_function* | Calculate sensitivity across all predictions, using the top alert_rate percent of observations as the predicted positive class. |
| *optimize_threshold* | Learn a binary classification threshold which optimizes the current objective. |
| *positive_only* | If True, this objective is only valid for positive data. Defaults to False. |
| *score* | Returns a numerical score indicating performance based on the differences between the predicted and actual values. |
| *validate_inputs* | Validate inputs for scoring. |

**classmethod calculate_percent_difference**(*cls*, *score*, *baseline_score*)

Calculate the percent difference between scores.

**Parameters**

- **score** (*float*) – A score. Output of the score method of this objective.
- **baseline_score** (*float*) – A score. Output of the score method of this objective. In practice, this is the score achieved on this objective with a baseline estimator.

**Returns**

**The percent difference between the scores. Note that for objectives that can be interpreted** as percentages, this will be the difference between the reference score and score. For all other objectives, the difference will be normalized by the reference score.

**Return type** float

**property can_optimize_threshold**(*cls*)

Returns a boolean determining if we can optimize the binary classification objective threshold.

This will be false for any objective that works directly with predicted probabilities, like log loss and AUC. Otherwise, it will be true.

**Returns** Whether or not an objective can be optimized.

**Return type** bool

`decision_function`(*self*, *ypred_proba*, ***kwargs*)

Determine if an observation is high risk given an alert rate.

**Parameters**

- **ypred_proba** (*pd.Series*) – Predicted probabilities.
- ****kwargs** – Additional abritrary parameters.

**Returns** Whether or not an observation is high risk given an alert rate.

**Return type** pd.Series

`classmethod is_defined_for_problem_type`(*cls*, *problem_type*)

Returns whether or not an objective is defined for a problem type.

`objective_function`(*self*, *y_true*, *y_predicted*, ***kwargs*)

Calculate sensitivity across all predictions, using the top alert_rate percent of observations as the predicted positive class.

**Parameters**

- **y_true** (*pd.Series*) – True labels.
- **y_predicted** (*pd.Series*) – Predicted labels based on alert_rate.
- ****kwargs** – Additional abritrary parameters.

**Returns** sensitivity using the observations with the top scores as the predicted positive class.

**Return type** float

`optimize_threshold`(*self*, *ypred_proba*, *y_true*, *X=None*)

Learn a binary classification threshold which optimizes the current objective.

**Parameters**

- **ypred_proba** (*pd.Series*) – The classifier's predicted probabilities
- **y_true** (*pd.Series*) – The ground truth for the predictions.
- **X** (*pd.DataFrame, optional*) – Any extra columns that are needed from training data.

**Returns** Optimal threshold for this objective.

**Raises** **RuntimeError** – If objective cannot be optimized.

`positive_only`(*cls*)

If True, this objective is only valid for positive data. Defaults to False.

`score`(*self*, *y_true*, *y_predicted*, *X=None*, *sample_weight=None*)

Returns a numerical score indicating performance based on the differences between the predicted and actual values.

**Parameters**

- **y_predicted** (*pd.Series*) – Predicted values of length [n_samples]
- **y_true** (*pd.Series*) – Actual class labels of length [n_samples]

- **X** (*pd.DataFrame or np.ndarray*) – Extra data of shape [n_samples, n_features] necessary to calculate score
- **sample_weight** (*pd.DataFrame or np.ndarray*) – Sample weights used in computing objective value result

**Returns** score

**validate_inputs**(*self*, *y_true*, *y_predicted*)

Validate inputs for scoring.

### standard_metrics

Standard machine learning objective functions.

### Module Contents

### Classes Summary

| | |
|---|---|
| *AccuracyBinary* | Accuracy score for binary classification. |
| *AccuracyMulticlass* | Accuracy score for multiclass classification. |
| *AUC* | AUC score for binary classification. |
| *AUCMacro* | AUC score for multiclass classification using macro averaging. |
| *AUCMicro* | AUC score for multiclass classification using micro averaging. |
| *AUCWeighted* | AUC Score for multiclass classification using weighted averaging. |
| *BalancedAccuracyBinary* | Balanced accuracy score for binary classification. |
| *BalancedAccuracyMulticlass* | Balanced accuracy score for multiclass classification. |
| *ExpVariance* | Explained variance score for regression. |
| *F1* | F1 score for binary classification. |
| *F1Macro* | F1 score for multiclass classification using macro averaging. |
| *F1Micro* | F1 score for multiclass classification using micro averaging. |
| *F1Weighted* | F1 score for multiclass classification using weighted averaging. |
| *Gini* | Gini coefficient for binary classification. |
| *LogLossBinary* | Log Loss for binary classification. |
| *LogLossMulticlass* | Log Loss for multiclass classification. |
| *MAE* | Mean absolute error for regression. |
| *MAPE* | Mean absolute percentage error for time series regression. Scaled by 100 to return a percentage. |
| *MaxError* | Maximum residual error for regression. |
| *MCCBinary* | Matthews correlation coefficient for binary classification. |
| *MCCMulticlass* | Matthews correlation coefficient for multiclass classification. |
| *MeanSquaredLogError* | Mean squared log error for regression. |

continues on next page

Table 3 – continued from previous page

| | |
|---|---|
| *MedianAE* | Median absolute error for regression. |
| *MSE* | Mean squared error for regression. |
| *Precision* | Precision score for binary classification. |
| *PrecisionMacro* | Precision score for multiclass classification using macro-averaging. |
| *PrecisionMicro* | Precision score for multiclass classification using micro averaging. |
| *PrecisionWeighted* | Precision score for multiclass classification using weighted averaging. |
| *R2* | Coefficient of determination for regression. |
| *Recall* | Recall score for binary classification. |
| *RecallMacro* | Recall score for multiclass classification using macro averaging. |
| *RecallMicro* | Recall score for multiclass classification using micro averaging. |
| *RecallWeighted* | Recall score for multiclass classification using weighted averaging. |
| *RootMeanSquaredError* | Root mean squared error for regression. |
| *RootMeanSquaredLogError* | Root mean squared log error for regression. |

## Contents

**class** evalml.objectives.standard_metrics.**AccuracyBinary**

Accuracy score for binary classification.

### Example

```
>>> y_true = pd.Series([0, 0, 0, 0, 0, 1, 1, 1, 0, 1, 1])
>>> y_pred = pd.Series([0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1])
>>> np.testing.assert_almost_equal(AccuracyBinary().objective_function(y_true, y_
↪pred), 0.6363636)
```

**Attributes**

| | |
|---|---|
| **expected_range** | [0, 1] |
| **greater_is_better** | True |
| **is_bounded_like_percentage** | True |
| **name** | Accuracy Binary |
| **perfect_score** | 1.0 |
| **problem_types** | [ProblemTypes.BINARY, ProblemTypes.TIME_SERIES_BINARY] |
| **score_needs_proba** | False |

**Methods**

| | |
|---|---|
| *calculate_percent_difference* | Calculate the percent difference between scores. |
| *can_optimize_threshold* | Returns a boolean determining if we can optimize the binary classification objective threshold. |
| *decision_function* | Apply a learned threshold to predicted probabilities to get predicted classes. |
| *is_defined_for_problem_type* | Returns whether or not an objective is defined for a problem type. |
| *objective_function* | Objective function for accuracy score for binary classification. |
| *optimize_threshold* | Learn a binary classification threshold which optimizes the current objective. |
| *positive_only* | If True, this objective is only valid for positive data. Defaults to False. |
| *score* | Returns a numerical score indicating performance based on the differences between the predicted and actual values. |
| *validate_inputs* | Validate inputs for scoring. |

**classmethod calculate_percent_difference**(*cls*, *score*, *baseline_score*)

Calculate the percent difference between scores.

**Parameters**

- **score** (*float*) – A score. Output of the score method of this objective.
- **baseline_score** (*float*) – A score. Output of the score method of this objective. In practice, this is the score achieved on this objective with a baseline estimator.

**Returns**

**The percent difference between the scores. Note that for objectives that can be interpreted** as percentages, this will be the difference between the reference score and score. For all other objectives, the difference will be normalized by the reference score.

**Return type** float

**property can_optimize_threshold**(*cls*)

Returns a boolean determining if we can optimize the binary classification objective threshold.

This will be false for any objective that works directly with predicted probabilities, like log loss and AUC. Otherwise, it will be true.

**Returns** Whether or not an objective can be optimized.

**Return type** bool

**decision_function**(*self*, *ypred_proba*, *threshold=0.5*, *X=None*)

Apply a learned threshold to predicted probabilities to get predicted classes.

**Parameters**

- **ypred_proba** (*pd.Series, np.ndarray*) – The classifier's predicted probabilities
- **threshold** (*float, optional*) – Threshold used to make a prediction. Defaults to 0.5.
- **X** (*pd.DataFrame, optional*) – Any extra columns that are needed from training data.

**Returns** predictions

**classmethod is_defined_for_problem_type**(*cls*, *problem_type*)

Returns whether or not an objective is defined for a problem type.

**objective_function**(*self*, *y_true*, *y_predicted*, *X=None*, *sample_weight=None*)

Objective function for accuracy score for binary classification.

**optimize_threshold**(*self*, *ypred_proba*, *y_true*, *X=None*)

Learn a binary classification threshold which optimizes the current objective.

**Parameters**

- **ypred_proba** (*pd.Series*) – The classifier's predicted probabilities
- **y_true** (*pd.Series*) – The ground truth for the predictions.
- **X** (*pd.DataFrame, optional*) – Any extra columns that are needed from training data.

**Returns** Optimal threshold for this objective.

**Raises** **RuntimeError** – If objective cannot be optimized.

**positive_only**(*cls*)

If True, this objective is only valid for positive data. Defaults to False.

**score**(*self*, *y_true*, *y_predicted*, *X=None*, *sample_weight=None*)

Returns a numerical score indicating performance based on the differences between the predicted and actual values.

**Parameters**

- **y_predicted** (*pd.Series*) – Predicted values of length [n_samples]
- **y_true** (*pd.Series*) – Actual class labels of length [n_samples]
- **X** (*pd.DataFrame or np.ndarray*) – Extra data of shape [n_samples, n_features] necessary to calculate score
- **sample_weight** (*pd.DataFrame or np.ndarray*) – Sample weights used in computing objective value result

**Returns** score

**validate_inputs**(*self*, *y_true*, *y_predicted*)

Validate inputs for scoring.

**class** evalml.objectives.standard_metrics.**AccuracyMulticlass**

Accuracy score for multiclass classification.

### Example

```
>>> y_true = pd.Series([0, 1, 0, 2, 0, 1, 2, 1, 2, 0, 2])
>>> y_pred = pd.Series([0, 0, 0, 0, 1, 1, 1, 1, 2, 2, 2])
>>> np.testing.assert_almost_equal(AccuracyMulticlass().objective_function(y_true,␣
↪y_pred), 0.5454545)
```

**Attributes**

| | |
|---|---|
| **expected_range** | [0, 1] |
| **greater_is_better** | True |
| **is_bounded_like** | True |
| **name** | Accuracy Multiclass |
| **perfect_score** | 1.0 |
| **problem_types** | [ProblemTypes.MULTICLASS, ProblemTypes.TIME_SERIES_MULTICLASS] |
| **score_needs_proba** | False |

**Methods**

| | |
|---|---|
| *calculate_percent_difference* | Calculate the percent difference between scores. |
| *is_defined_for_problem_type* | Returns whether or not an objective is defined for a problem type. |
| *objective_function* | Objective function for accuracy score for multiclass classification. |
| *positive_only* | If True, this objective is only valid for positive data. Defaults to False. |
| *score* | Returns a numerical score indicating performance based on the differences between the predicted and actual values. |
| *validate_inputs* | Validates the input based on a few simple checks. |

**classmethod calculate_percent_difference**(*cls*, *score*, *baseline_score*)

Calculate the percent difference between scores.

**Parameters**

- **score** (*float*) – A score. Output of the score method of this objective.
- **baseline_score** (*float*) – A score. Output of the score method of this objective. In practice, this is the score achieved on this objective with a baseline estimator.

**Returns**

**The percent difference between the scores. Note that for objectives that can be interpreted** as percentages, this will be the difference between the reference score and score. For all other objectives, the difference will be normalized by the reference score.

**Return type** float

**classmethod is_defined_for_problem_type**(*cls*, *problem_type*)

Returns whether or not an objective is defined for a problem type.

**objective_function**(*self*, *y_true*, *y_predicted*, *X=None*, *sample_weight=None*)

Objective function for accuracy score for multiclass classification.

**positive_only**(*cls*)

If True, this objective is only valid for positive data. Defaults to False.

**score**(*self*, *y_true*, *y_predicted*, *X=None*, *sample_weight=None*)

Returns a numerical score indicating performance based on the differences between the predicted and actual values.

**Parameters**

- **y_predicted** (*pd.Series*) – Predicted values of length [n_samples]
- **y_true** (*pd.Series*) – Actual class labels of length [n_samples]
- **X** (*pd.DataFrame or np.ndarray*) – Extra data of shape [n_samples, n_features] necessary to calculate score
- **sample_weight** (*pd.DataFrame or np.ndarray*) – Sample weights used in computing objective value result

**Returns** score

**validate_inputs**(*self*, *y_true*, *y_predicted*)

Validates the input based on a few simple checks.

**Parameters**

- **y_predicted** (*pd.Series, or pd.DataFrame*) – Predicted values of length [n_samples].
- **y_true** (*pd.Series*) – Actual class labels of length [n_samples].

**Raises** **ValueError** – If the inputs are malformed.

**class** evalml.objectives.standard_metrics.**AUC**

AUC score for binary classification.

### Example

```
>>> y_true = pd.Series([0, 1, 1, 0, 0, 1, 1, 1, 0, 1, 1])
>>> y_pred = pd.Series([0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1])
>>> np.testing.assert_almost_equal(AUC().objective_function(y_true, y_pred), 0.
↪5714285)
```

**Attributes**

| | |
|---|---|
| **expected_range** | [0, 1] |
| **greater_is_better** | True |
| **is_bounded_likely** | True |
| **name** | AUC |
| **perfect_score** | 1.0 |
| **problem_types** | [ProblemTypes.BINARY, ProblemTypes.TIME_SERIES_BINARY] |
| **score_needs_proba** | True |

**Methods**

| | |
|---|---|
| *calculate_percent_difference* | Calculate the percent difference between scores. |
| *can_optimize_threshold* | Returns a boolean determining if we can optimize the binary classification objective threshold. |
| *decision_function* | Apply a learned threshold to predicted probabilities to get predicted classes. |
| *is_defined_for_problem_type* | Returns whether or not an objective is defined for a problem type. |
| *objective_function* | Objective function for AUC score for binary classification. |
| *optimize_threshold* | Learn a binary classification threshold which optimizes the current objective. |
| *positive_only* | If True, this objective is only valid for positive data. Defaults to False. |
| *score* | Returns a numerical score indicating performance based on the differences between the predicted and actual values. |
| *validate_inputs* | Validate inputs for scoring. |

**classmethod calculate_percent_difference**(*cls*, *score*, *baseline_score*)

Calculate the percent difference between scores.

**Parameters**

- **score** (*float*) – A score. Output of the score method of this objective.
- **baseline_score** (*float*) – A score. Output of the score method of this objective. In practice, this is the score achieved on this objective with a baseline estimator.

**Returns**

**The percent difference between the scores. Note that for objectives that can be interpreted** as percentages, this will be the difference between the reference score and score. For all other objectives, the difference will be normalized by the reference score.

**Return type** float

**property can_optimize_threshold**(*cls*)

Returns a boolean determining if we can optimize the binary classification objective threshold.

This will be false for any objective that works directly with predicted probabilities, like log loss and AUC. Otherwise, it will be true.

**Returns** Whether or not an objective can be optimized.

**Return type** bool

**decision_function**(*self*, *ypred_proba*, *threshold=0.5*, *X=None*)

Apply a learned threshold to predicted probabilities to get predicted classes.

**Parameters**

- **ypred_proba** (*pd.Series, np.ndarray*) – The classifier's predicted probabilities
- **threshold** (*float, optional*) – Threshold used to make a prediction. Defaults to 0.5.
- **X** (*pd.DataFrame, optional*) – Any extra columns that are needed from training data.

**Returns** predictions

**classmethod is_defined_for_problem_type**(*cls*, *problem_type*)

Returns whether or not an objective is defined for a problem type.

**objective_function**(*self*, *y_true*, *y_predicted*, *X=None*, *sample_weight=None*)

Objective function for AUC score for binary classification.

**optimize_threshold**(*self*, *ypred_proba*, *y_true*, *X=None*)

Learn a binary classification threshold which optimizes the current objective.

**Parameters**

- **ypred_proba** (*pd.Series*) – The classifier's predicted probabilities
- **y_true** (*pd.Series*) – The ground truth for the predictions.
- **X** (*pd.DataFrame, optional*) – Any extra columns that are needed from training data.

**Returns** Optimal threshold for this objective.

**Raises** **RuntimeError** – If objective cannot be optimized.

**positive_only**(*cls*)

If True, this objective is only valid for positive data. Defaults to False.

**score**(*self*, *y_true*, *y_predicted*, *X=None*, *sample_weight=None*)

Returns a numerical score indicating performance based on the differences between the predicted and actual values.

**Parameters**

- **y_predicted** (*pd.Series*) – Predicted values of length [n_samples]
- **y_true** (*pd.Series*) – Actual class labels of length [n_samples]
- **X** (*pd.DataFrame or np.ndarray*) – Extra data of shape [n_samples, n_features] necessary to calculate score
- **sample_weight** (*pd.DataFrame or np.ndarray*) – Sample weights used in computing objective value result

**Returns** score

**validate_inputs**(*self*, *y_true*, *y_predicted*)

Validate inputs for scoring.

**class** evalml.objectives.standard_metrics.**AUCMacro**

AUC score for multiclass classification using macro averaging.

### Example

```
>>> y_true = [0, 1, 2, 0, 2, 1]
>>> y_pred = [[0.7, 0.2, 0.1],
...           [0.1, 0.0, 0.9],
...           [0.1, 0.3, 0.6],
...           [0.9, 0.1, 0.0],
...           [0.6, 0.1, 0.3],
...           [0.5, 0.5, 0.0]]
>>> np.testing.assert_almost_equal(AUCMacro().objective_function(y_true, y_pred), 0.
↪75)
```

**Attributes**

| | |
|---|---|
| **expected_range** | [0, 1] |
| **greater_is_better** | True |
| **is_bounded_like_percentage** | True |
| **name** | AUC Macro |
| **perfect_score** | 1.0 |
| **problem_types** | [ProblemTypes.MULTICLASS, ProblemTypes.TIME_SERIES_MULTICLASS] |
| **score_needs_proba** | True |

**Methods**

| | |
|---|---|
| *calculate_percent_difference* | Calculate the percent difference between scores. |
| *is_defined_for_problem_type* | Returns whether or not an objective is defined for a problem type. |
| *objective_function* | Objective function for AUC score for multiclass classification using macro-averaging. |
| *positive_only* | If True, this objective is only valid for positive data. Defaults to False. |
| *score* | Returns a numerical score indicating performance based on the differences between the predicted and actual values. |
| *validate_inputs* | Validates the input based on a few simple checks. |

**classmethod calculate_percent_difference**(*cls*, *score*, *baseline_score*)

Calculate the percent difference between scores.

**Parameters**

- **score** (*float*) – A score. Output of the score method of this objective.
- **baseline_score** (*float*) – A score. Output of the score method of this objective. In practice, this is the score achieved on this objective with a baseline estimator.

**Returns**

**The percent difference between the scores. Note that for objectives that can be interpreted** as percentages, this will be the difference between the reference score and score. For all other objectives, the difference will be normalized by the reference score.

**Return type** float

**classmethod is_defined_for_problem_type**(*cls*, *problem_type*)

Returns whether or not an objective is defined for a problem type.

**objective_function**(*self*, *y_true*, *y_predicted*, *X=None*, *sample_weight=None*)

Objective function for AUC score for multiclass classification using macro-averaging.

**positive_only**(*cls*)

If True, this objective is only valid for positive data. Defaults to False.

**score**(*self*, *y_true*, *y_predicted*, *X=None*, *sample_weight=None*)

Returns a numerical score indicating performance based on the differences between the predicted and actual values.

**Parameters**

- **y_predicted** (*pd.Series*) – Predicted values of length [n_samples]
- **y_true** (*pd.Series*) – Actual class labels of length [n_samples]
- **X** (*pd.DataFrame or np.ndarray*) – Extra data of shape [n_samples, n_features] necessary to calculate score
- **sample_weight** (*pd.DataFrame or np.ndarray*) – Sample weights used in computing objective value result

**Returns** score

**validate_inputs**(*self*, *y_true*, *y_predicted*)

Validates the input based on a few simple checks.

**Parameters**

- **y_predicted** (*pd.Series, or pd.DataFrame*) – Predicted values of length [n_samples].
- **y_true** (*pd.Series*) – Actual class labels of length [n_samples].

**Raises** **ValueError** – If the inputs are malformed.

**class** evalml.objectives.standard_metrics.**AUCMicro**

AUC score for multiclass classification using micro averaging.

### Example

```
>>> y_true = [0, 1, 2, 0, 2, 1]
>>> y_pred = [[0.7, 0.2, 0.1],
...           [0.3, 0.5, 0.2],
...           [0.1, 0.3, 0.6],
...           [0.9, 0.1, 0.0],
...           [0.3, 0.1, 0.6],
...           [0.5, 0.5, 0.0]]
>>> np.testing.assert_almost_equal(AUCMicro().objective_function(y_true, y_pred), 0.
↪9861111)
```

**Attributes**

| | |
|---|---|
| **expected_range** | [0, 1] |
| **greater_is_better** | True |
| **is_bounded_likely** | True |
| **name** | AUC Micro |
| **perfect_score** | 1.0 |
| **problem_types** | [ProblemTypes.MULTICLASS, ProblemTypes.TIME_SERIES_MULTICLASS] |
| **score_needs_proba** | True |

**Methods**

| | |
|---|---|
| *calculate_percent_difference* | Calculate the percent difference between scores. |
| *is_defined_for_problem_type* | Returns whether or not an objective is defined for a problem type. |
| *objective_function* | Objective function for AUC score for multiclass classification using micro-averaging. |
| *positive_only* | If True, this objective is only valid for positive data. Defaults to False. |
| *score* | Returns a numerical score indicating performance based on the differences between the predicted and actual values. |
| *validate_inputs* | Validates the input based on a few simple checks. |

**classmethod calculate_percent_difference**(*cls*, *score*, *baseline_score*)

Calculate the percent difference between scores.

**Parameters**

- **score** (*float*) – A score. Output of the score method of this objective.
- **baseline_score** (*float*) – A score. Output of the score method of this objective. In practice, this is the score achieved on this objective with a baseline estimator.

**Returns**

**The percent difference between the scores. Note that for objectives that can be interpreted** as percentages, this will be the difference between the reference score and score. For all other objectives, the difference will be normalized by the reference score.

**Return type** float

**classmethod is_defined_for_problem_type**(*cls*, *problem_type*)

Returns whether or not an objective is defined for a problem type.

**objective_function**(*self*, *y_true*, *y_predicted*, *X=None*, *sample_weight=None*)

Objective function for AUC score for multiclass classification using micro-averaging.

**positive_only**(*cls*)

If True, this objective is only valid for positive data. Defaults to False.

**score**(*self*, *y_true*, *y_predicted*, *X=None*, *sample_weight=None*)

Returns a numerical score indicating performance based on the differences between the predicted and actual values.

**Parameters**

- **y_predicted** (*pd.Series*) – Predicted values of length [n_samples]
- **y_true** (*pd.Series*) – Actual class labels of length [n_samples]
- **X** (*pd.DataFrame or np.ndarray*) – Extra data of shape [n_samples, n_features] necessary to calculate score
- **sample_weight** (*pd.DataFrame or np.ndarray*) – Sample weights used in computing objective value result

**Returns** score

**validate_inputs**(*self*, *y_true*, *y_predicted*)

Validates the input based on a few simple checks.

**Parameters**

- **y_predicted** (*pd.Series, or pd.DataFrame*) – Predicted values of length [n_samples].
- **y_true** (*pd.Series*) – Actual class labels of length [n_samples].

**Raises** **ValueError** – If the inputs are malformed.

**class** evalml.objectives.standard_metrics.**AUCWeighted**

AUC Score for multiclass classification using weighted averaging.

### Example

```
>>> y_true = [0, 1, 2, 0, 2, 1]
>>> y_pred = [[0.7, 0.2, 0.1],
...           [0.1, 0.0, 0.9],
...           [0.1, 0.3, 0.6],
...           [0.1, 0.2, 0.7],
...           [0.6, 0.1, 0.3],
...           [0.5, 0.2, 0.3]]
>>> np.testing.assert_almost_equal(AUCWeighted().objective_function(y_true, y_pred),
↪ 0.4375)
```

**Attributes**

| | |
|---|---|
| **expected_range** | [0, 1] |
| **greater_is_better** | True |
| **is_bounded_like** | True |
| **name** | AUC Weighted |
| **perfect_score** | 1.0 |
| **problem_types** | [ProblemTypes.MULTICLASS, ProblemTypes.TIME_SERIES_MULTICLASS] |
| **score_needs_proba** | True |

**Methods**

| | |
|---|---|
| *calculate_percent_difference* | Calculate the percent difference between scores. |
| *is_defined_for_problem_type* | Returns whether or not an objective is defined for a problem type. |
| *objective_function* | Objective function for AUC Score for multiclass classification using weighted averaging. |
| *positive_only* | If True, this objective is only valid for positive data. Defaults to False. |
| *score* | Returns a numerical score indicating performance based on the differences between the predicted and actual values. |
| *validate_inputs* | Validates the input based on a few simple checks. |

**classmethod calculate_percent_difference**(*cls*, *score*, *baseline_score*)

Calculate the percent difference between scores.

**Parameters**

- **score** (*float*) – A score. Output of the score method of this objective.
- **baseline_score** (*float*) – A score. Output of the score method of this objective. In practice, this is the score achieved on this objective with a baseline estimator.

**Returns**

**The percent difference between the scores. Note that for objectives that can be interpreted** as percentages, this will be the difference between the reference score and score. For all other objectives, the difference will be normalized by the reference score.

**Return type** float

**classmethod is_defined_for_problem_type**(*cls*, *problem_type*)

Returns whether or not an objective is defined for a problem type.

**objective_function**(*self*, *y_true*, *y_predicted*, *X=None*, *sample_weight=None*)

Objective function for AUC Score for multiclass classification using weighted averaging.

**positive_only**(*cls*)

If True, this objective is only valid for positive data. Defaults to False.

**score**(*self*, *y_true*, *y_predicted*, *X=None*, *sample_weight=None*)

Returns a numerical score indicating performance based on the differences between the predicted and actual values.

**Parameters**

- **y_predicted** (*pd.Series*) – Predicted values of length [n_samples]
- **y_true** (*pd.Series*) – Actual class labels of length [n_samples]
- **X** (*pd.DataFrame or np.ndarray*) – Extra data of shape [n_samples, n_features] necessary to calculate score
- **sample_weight** (*pd.DataFrame or np.ndarray*) – Sample weights used in computing objective value result

**Returns** score

**validate_inputs**(*self*, *y_true*, *y_predicted*)

Validates the input based on a few simple checks.

**Parameters**

- **y_predicted** (*pd.Series, or pd.DataFrame*) – Predicted values of length [n_samples].
- **y_true** (*pd.Series*) – Actual class labels of length [n_samples].

**Raises** **ValueError** – If the inputs are malformed.

**class** evalml.objectives.standard_metrics.**BalancedAccuracyBinary**

Balanced accuracy score for binary classification.

### Example

```
>>> y_true = pd.Series([0, 0, 0, 0, 0, 1, 1, 1, 0, 1, 1])
>>> y_pred = pd.Series([0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1])
>>> np.testing.assert_almost_equal(BalancedAccuracyBinary().objective_function(y_
↪true, y_pred), 0.60)
```

**Attributes**

| | |
|---|---|
| **expected_range** | [0, 1] |
| **greater_is_better** | True |
| **is_bounded_like_percentage** | True |
| **name** | Balanced Accuracy Binary |
| **perfect_score** | 1.0 |
| **problem_types** | [ProblemTypes.BINARY, ProblemTypes.TIME_SERIES_BINARY] |
| **score_needs_proba** | False |

**Methods**

| | |
|---|---|
| *calculate_percent_difference* | Calculate the percent difference between scores. |
| *can_optimize_threshold* | Returns a boolean determining if we can optimize the binary classification objective threshold. |
| *decision_function* | Apply a learned threshold to predicted probabilities to get predicted classes. |
| *is_defined_for_problem_type* | Returns whether or not an objective is defined for a problem type. |
| *objective_function* | Objective function for accuracy score for balanced accuracy for binary classification. |
| *optimize_threshold* | Learn a binary classification threshold which optimizes the current objective. |
| *positive_only* | If True, this objective is only valid for positive data. Defaults to False. |
| *score* | Returns a numerical score indicating performance based on the differences between the predicted and actual values. |
| *validate_inputs* | Validate inputs for scoring. |

**classmethod calculate_percent_difference**(*cls*, *score*, *baseline_score*)

Calculate the percent difference between scores.

**Parameters**

- **score** (*float*) – A score. Output of the score method of this objective.
- **baseline_score** (*float*) – A score. Output of the score method of this objective. In practice, this is the score achieved on this objective with a baseline estimator.

**Returns**

**The percent difference between the scores. Note that for objectives that can be interpreted** as percentages, this will be the difference between the reference score and score. For all other objectives, the difference will be normalized by the reference score.

**Return type** float

**property can_optimize_threshold**(*cls*)

Returns a boolean determining if we can optimize the binary classification objective threshold.

This will be false for any objective that works directly with predicted probabilities, like log loss and AUC. Otherwise, it will be true.

**Returns** Whether or not an objective can be optimized.

**Return type** bool

**decision_function**(*self*, *ypred_proba*, *threshold=0.5*, *X=None*)

Apply a learned threshold to predicted probabilities to get predicted classes.

**Parameters**

- **ypred_proba** (*pd.Series, np.ndarray*) – The classifier's predicted probabilities
- **threshold** (*float, optional*) – Threshold used to make a prediction. Defaults to 0.5.
- **X** (*pd.DataFrame, optional*) – Any extra columns that are needed from training data.

**Returns** predictions

**classmethod is_defined_for_problem_type**(*cls*, *problem_type*)

Returns whether or not an objective is defined for a problem type.

**objective_function**(*self*, *y_true*, *y_predicted*, *X=None*, *sample_weight=None*)

Objective function for accuracy score for balanced accuracy for binary classification.

**optimize_threshold**(*self*, *ypred_proba*, *y_true*, *X=None*)

Learn a binary classification threshold which optimizes the current objective.

**Parameters**

- **ypred_proba** (*pd.Series*) – The classifier's predicted probabilities
- **y_true** (*pd.Series*) – The ground truth for the predictions.
- **X** (*pd.DataFrame, optional*) – Any extra columns that are needed from training data.

**Returns** Optimal threshold for this objective.

**Raises** **RuntimeError** – If objective cannot be optimized.

**positive_only**(*cls*)

If True, this objective is only valid for positive data. Defaults to False.

**score**(*self*, *y_true*, *y_predicted*, *X=None*, *sample_weight=None*)

Returns a numerical score indicating performance based on the differences between the predicted and actual values.

**Parameters**

- **y_predicted** (*pd.Series*) – Predicted values of length [n_samples]
- **y_true** (*pd.Series*) – Actual class labels of length [n_samples]
- **X** (*pd.DataFrame or np.ndarray*) – Extra data of shape [n_samples, n_features] necessary to calculate score
- **sample_weight** (*pd.DataFrame or np.ndarray*) – Sample weights used in computing objective value result

**Returns** score

**validate_inputs**(*self*, *y_true*, *y_predicted*)

Validate inputs for scoring.

**class** evalml.objectives.standard_metrics.**BalancedAccuracyMulticlass**

Balanced accuracy score for multiclass classification.

### Example

```
>>> y_true = pd.Series([0, 1, 0, 2, 0, 1, 2, 1, 2, 0, 2])
>>> y_pred = pd.Series([0, 0, 0, 0, 1, 1, 1, 1, 2, 2, 2])
>>> np.testing.assert_almost_equal(BalancedAccuracyMulticlass().objective_
↪function(y_true, y_pred), 0.5555555)
```

**Attributes**

| | |
|---|---|
| **expected_range** | [0, 1] |
| **greater_is_better** | True |
| **is_bounded_like** | True |
| **name** | Balanced Accuracy Multiclass |
| **perfect_score** | 1.0 |
| **problem_types** | [ProblemTypes.MULTICLASS, ProblemTypes.TIME_SERIES_MULTICLASS] |
| **score_needs_proba** | False |

**Methods**

| | |
|---|---|
| *calculate_percent_difference* | Calculate the percent difference between scores. |
| *is_defined_for_problem_type* | Returns whether or not an objective is defined for a problem type. |
| *objective_function* | Objective function for accuracy score for balanced accuracy for multiclass classification. |
| *positive_only* | If True, this objective is only valid for positive data. Defaults to False. |
| *score* | Returns a numerical score indicating performance based on the differences between the predicted and actual values. |
| *validate_inputs* | Validates the input based on a few simple checks. |

**classmethod calculate_percent_difference**(*cls*, *score*, *baseline_score*)

Calculate the percent difference between scores.

**Parameters**

- **score** (*float*) – A score. Output of the score method of this objective.
- **baseline_score** (*float*) – A score. Output of the score method of this objective. In practice, this is the score achieved on this objective with a baseline estimator.

**Returns**

**The percent difference between the scores. Note that for objectives that can be interpreted** as percentages, this will be the difference between the reference score and score. For all other objectives, the difference will be normalized by the reference score.

**Return type** float

**classmethod is_defined_for_problem_type**(*cls*, *problem_type*)

Returns whether or not an objective is defined for a problem type.

**objective_function**(*self*, *y_true*, *y_predicted*, *X=None*, *sample_weight=None*)

Objective function for accuracy score for balanced accuracy for multiclass classification.

**positive_only**(*cls*)

If True, this objective is only valid for positive data. Defaults to False.

**score**(*self*, *y_true*, *y_predicted*, *X=None*, *sample_weight=None*)

Returns a numerical score indicating performance based on the differences between the predicted and actual values.

**Parameters**

- **y_predicted** (*pd.Series*) – Predicted values of length [n_samples]
- **y_true** (*pd.Series*) – Actual class labels of length [n_samples]
- **X** (*pd.DataFrame or np.ndarray*) – Extra data of shape [n_samples, n_features] necessary to calculate score
- **sample_weight** (*pd.DataFrame or np.ndarray*) – Sample weights used in computing objective value result

**Returns** score

**validate_inputs**(*self*, *y_true*, *y_predicted*)

Validates the input based on a few simple checks.

**Parameters**

- **y_predicted** (*pd.Series, or pd.DataFrame*) – Predicted values of length [n_samples].
- **y_true** (*pd.Series*) – Actual class labels of length [n_samples].

**Raises** **ValueError** – If the inputs are malformed.

**class** evalml.objectives.standard_metrics.**ExpVariance**

Explained variance score for regression.

### Example

```
>>> y_true = pd.Series([1.5, 2, 3, 1, 0.5, 1, 2.5, 2.5, 1, 0.5, 2])
>>> y_pred = pd.Series([1.5, 2.5, 2, 1, 0.5, 1, 3, 2.25, 0.75, 0.25, 1.75])
>>> np.testing.assert_almost_equal(ExpVariance().objective_function(y_true, y_pred),
↪ 0.7760736)
```

**Attributes**

| | |
|---|---|
| **expected_range** | None |
| **greater_is_better** | True |
| **is_bounded_like_percentage** | False |
| **name** | ExpVariance |
| **perfect_score** | 1.0 |
| **problem_types** | [ProblemTypes.REGRESSION, ProblemTypes.TIME_SERIES_REGRESSION] |
| **score_needs_proba** | False |

**Methods**

| | |
|---|---|
| *calculate_percent_difference* | Calculate the percent difference between scores. |
| *is_defined_for_problem_type* | Returns whether or not an objective is defined for a problem type. |
| *objective_function* | Objective function for explained variance score for regression. |
| *positive_only* | If True, this objective is only valid for positive data. Defaults to False. |
| *score* | Returns a numerical score indicating performance based on the differences between the predicted and actual values. |
| *validate_inputs* | Validates the input based on a few simple checks. |

**classmethod calculate_percent_difference**(*cls*, *score*, *baseline_score*)

Calculate the percent difference between scores.

**Parameters**

- **score** (*float*) – A score. Output of the score method of this objective.
- **baseline_score** (*float*) – A score. Output of the score method of this objective. In practice, this is the score achieved on this objective with a baseline estimator.

**Returns**

**The percent difference between the scores. Note that for objectives that can be interpreted** as percentages, this will be the difference between the reference score and score. For all other objectives, the difference will be normalized by the reference score.

**Return type** float

**classmethod is_defined_for_problem_type**(*cls*, *problem_type*)

Returns whether or not an objective is defined for a problem type.

**objective_function**(*self*, *y_true*, *y_predicted*, *X=None*, *sample_weight=None*)

Objective function for explained variance score for regression.

**positive_only**(*cls*)

If True, this objective is only valid for positive data. Defaults to False.

**score**(*self*, *y_true*, *y_predicted*, *X=None*, *sample_weight=None*)

Returns a numerical score indicating performance based on the differences between the predicted and actual values.

**Parameters**

- **y_predicted** (*pd.Series*) – Predicted values of length [n_samples]
- **y_true** (*pd.Series*) – Actual class labels of length [n_samples]
- **X** (*pd.DataFrame or np.ndarray*) – Extra data of shape [n_samples, n_features] necessary to calculate score
- **sample_weight** (*pd.DataFrame or np.ndarray*) – Sample weights used in computing objective value result

**Returns** score

**validate_inputs**(*self*, *y_true*, *y_predicted*)

Validates the input based on a few simple checks.

**Parameters**

- **y_predicted** (*pd.Series, or pd.DataFrame*) – Predicted values of length [n_samples].
- **y_true** (*pd.Series*) – Actual class labels of length [n_samples].

**Raises** **ValueError** – If the inputs are malformed.

**class** evalml.objectives.standard_metrics.**F1**

F1 score for binary classification.

### Example

```
>>> y_true = pd.Series([0, 1, 1, 0, 0, 1, 1, 1, 0, 1, 1])
>>> y_pred = pd.Series([0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1])
>>> np.testing.assert_almost_equal(F1().objective_function(y_true, y_pred), 0.25)
```

**Attributes**

| | |
|---|---|
| **expected_range** | [0, 1] |
| **greater_is_better** | True |
| **is_bounded_like** | True |
| **name** | F1 |
| **perfect_score** | 1.0 |
| **problem_types** | [ProblemTypes.BINARY, ProblemTypes.TIME_SERIES_BINARY] |
| **score_needs_proba** | False |

**Methods**

| | |
|---|---|
| *calculate_percent_difference* | Calculate the percent difference between scores. |
| *can_optimize_threshold* | Returns a boolean determining if we can optimize the binary classification objective threshold. |
| *decision_function* | Apply a learned threshold to predicted probabilities to get predicted classes. |
| *is_defined_for_problem_type* | Returns whether or not an objective is defined for a problem type. |
| *objective_function* | Objective function for F1 score for binary classification. |
| *optimize_threshold* | Learn a binary classification threshold which optimizes the current objective. |
| *positive_only* | If True, this objective is only valid for positive data. Defaults to False. |
| *score* | Returns a numerical score indicating performance based on the differences between the predicted and actual values. |
| *validate_inputs* | Validate inputs for scoring. |

**classmethod calculate_percent_difference**(*cls*, *score*, *baseline_score*)

Calculate the percent difference between scores.

**Parameters**

- **score** (*float*) – A score. Output of the score method of this objective.
- **baseline_score** (*float*) – A score. Output of the score method of this objective. In practice, this is the score achieved on this objective with a baseline estimator.

**Returns**

**The percent difference between the scores. Note that for objectives that can be interpreted** as percentages, this will be the difference between the reference score and score. For all other objectives, the difference will be normalized by the reference score.

**Return type** float

**property can_optimize_threshold**(*cls*)

Returns a boolean determining if we can optimize the binary classification objective threshold.

This will be false for any objective that works directly with predicted probabilities, like log loss and AUC. Otherwise, it will be true.

**Returns** Whether or not an objective can be optimized.

**Return type** bool

**decision_function**(*self*, *ypred_proba*, *threshold=0.5*, *X=None*)

Apply a learned threshold to predicted probabilities to get predicted classes.

**Parameters**

- **ypred_proba** (*pd.Series, np.ndarray*) – The classifier's predicted probabilities
- **threshold** (*float, optional*) – Threshold used to make a prediction. Defaults to 0.5.
- **X** (*pd.DataFrame, optional*) – Any extra columns that are needed from training data.

**Returns** predictions

**classmethod is_defined_for_problem_type**(*cls*, *problem_type*)

Returns whether or not an objective is defined for a problem type.

**objective_function**(*self*, *y_true*, *y_predicted*, *X=None*, *sample_weight=None*)

Objective function for F1 score for binary classification.

**optimize_threshold**(*self*, *ypred_proba*, *y_true*, *X=None*)

Learn a binary classification threshold which optimizes the current objective.

**Parameters**

- **ypred_proba** (*pd.Series*) – The classifier's predicted probabilities
- **y_true** (*pd.Series*) – The ground truth for the predictions.
- **X** (*pd.DataFrame, optional*) – Any extra columns that are needed from training data.

**Returns** Optimal threshold for this objective.

**Raises** **RuntimeError** – If objective cannot be optimized.

**positive_only**(*cls*)

If True, this objective is only valid for positive data. Defaults to False.

**score**(*self*, *y_true*, *y_predicted*, *X=None*, *sample_weight=None*)

Returns a numerical score indicating performance based on the differences between the predicted and actual values.

**Parameters**

- **y_predicted** (*pd.Series*) – Predicted values of length [n_samples]
- **y_true** (*pd.Series*) – Actual class labels of length [n_samples]
- **X** (*pd.DataFrame or np.ndarray*) – Extra data of shape [n_samples, n_features] necessary to calculate score
- **sample_weight** (*pd.DataFrame or np.ndarray*) – Sample weights used in computing objective value result

**Returns** score

**validate_inputs**(*self*, *y_true*, *y_predicted*)

Validate inputs for scoring.

**class** evalml.objectives.standard_metrics.**F1Macro**

F1 score for multiclass classification using macro averaging.

### Example

```
>>> y_true = pd.Series([0, 1, 0, 2, 0, 1, 2, 1, 2, 0, 2])
>>> y_pred = pd.Series([0, 0, 0, 0, 1, 1, 1, 1, 2, 2, 2])
>>> np.testing.assert_almost_equal(F1Macro().objective_function(y_true, y_pred), 0.
↪5476190)
```

**Attributes**

| | |
|---|---|
| **expected_range** | [0, 1] |
| **greater_is_better** | True |
| **is_bounded_likeTrue** | **percentage** |
| **name** | F1 Macro |
| **perfect_score** | 1.0 |
| **problem_types** | [ProblemTypes.MULTICLASS, ProblemTypes.TIME_SERIES_MULTICLASS] |
| **score_needs_proba** | False |

**Methods**

| | |
|---|---|
| *calculate_percent_difference* | Calculate the percent difference between scores. |
| *is_defined_for_problem_type* | Returns whether or not an objective is defined for a problem type. |
| *objective_function* | Objective function for F1 score for multiclass classification using macro averaging. |
| *positive_only* | If True, this objective is only valid for positive data. Defaults to False. |
| *score* | Returns a numerical score indicating performance based on the differences between the predicted and actual values. |
| *validate_inputs* | Validates the input based on a few simple checks. |

**classmethod calculate_percent_difference**(*cls*, *score*, *baseline_score*)

Calculate the percent difference between scores.

**Parameters**

- **score** (*float*) – A score. Output of the score method of this objective.
- **baseline_score** (*float*) – A score. Output of the score method of this objective. In practice, this is the score achieved on this objective with a baseline estimator.

**Returns**

**The percent difference between the scores. Note that for objectives that can be interpreted** as percentages, this will be the difference between the reference score and score. For all other objectives, the difference will be normalized by the reference score.

**Return type** float

**classmethod is_defined_for_problem_type**(*cls*, *problem_type*)

Returns whether or not an objective is defined for a problem type.

**objective_function**(*self*, *y_true*, *y_predicted*, *X=None*, *sample_weight=None*)

Objective function for F1 score for multiclass classification using macro averaging.

**positive_only**(*cls*)

If True, this objective is only valid for positive data. Defaults to False.

**score**(*self*, *y_true*, *y_predicted*, *X=None*, *sample_weight=None*)

Returns a numerical score indicating performance based on the differences between the predicted and actual values.

**Parameters**

- **y_predicted** (*pd.Series*) – Predicted values of length [n_samples]
- **y_true** (*pd.Series*) – Actual class labels of length [n_samples]
- **X** (*pd.DataFrame or np.ndarray*) – Extra data of shape [n_samples, n_features] necessary to calculate score
- **sample_weight** (*pd.DataFrame or np.ndarray*) – Sample weights used in computing objective value result

**Returns** score

**validate_inputs**(*self*, *y_true*, *y_predicted*)

Validates the input based on a few simple checks.

**Parameters**

- **y_predicted** (*pd.Series, or pd.DataFrame*) – Predicted values of length [n_samples].
- **y_true** (*pd.Series*) – Actual class labels of length [n_samples].

**Raises** **ValueError** – If the inputs are malformed.

**class** evalml.objectives.standard_metrics.**F1Micro**

F1 score for multiclass classification using micro averaging.

### Example

```
>>> y_true = pd.Series([0, 1, 0, 2, 0, 1, 2, 1, 2, 0, 2])
>>> y_pred = pd.Series([0, 0, 0, 0, 1, 1, 1, 1, 2, 2, 2])
>>> np.testing.assert_almost_equal(F1Micro().objective_function(y_true, y_pred), 0.
↪5454545)
```

**Attributes**

| | |
|---|---|
| **expected_range** | [0, 1] |
| **greater_is_better** | True |
| **is_bounded_likert** | True |
| **name** | F1 Micro |
| **perfect_score** | 1.0 |
| **problem_types** | [ProblemTypes.MULTICLASS, ProblemTypes.TIME_SERIES_MULTICLASS] |
| **score_needs_proba** | False |

**Methods**

| | |
|---|---|
| *calculate_percent_difference* | Calculate the percent difference between scores. |
| *is_defined_for_problem_type* | Returns whether or not an objective is defined for a problem type. |
| *objective_function* | Objective function for F1 score for multiclass classification. |
| *positive_only* | If True, this objective is only valid for positive data. Defaults to False. |
| *score* | Returns a numerical score indicating performance based on the differences between the predicted and actual values. |
| *validate_inputs* | Validates the input based on a few simple checks. |

**classmethod calculate_percent_difference**(*cls*, *score*, *baseline_score*)

Calculate the percent difference between scores.

**Parameters**

- **score** (*float*) – A score. Output of the score method of this objective.
- **baseline_score** (*float*) – A score. Output of the score method of this objective. In practice, this is the score achieved on this objective with a baseline estimator.

**Returns**

**The percent difference between the scores. Note that for objectives that can be interpreted** as percentages, this will be the difference between the reference score and score. For all other objectives, the difference will be normalized by the reference score.

**Return type** float

**classmethod is_defined_for_problem_type**(*cls*, *problem_type*)

Returns whether or not an objective is defined for a problem type.

**objective_function**(*self*, *y_true*, *y_predicted*, *X=None*, *sample_weight=None*)

Objective function for F1 score for multiclass classification.

**positive_only**(*cls*)

If True, this objective is only valid for positive data. Defaults to False.

**score**(*self*, *y_true*, *y_predicted*, *X=None*, *sample_weight=None*)

Returns a numerical score indicating performance based on the differences between the predicted and actual values.

**Parameters**

- **y_predicted** (*pd.Series*) – Predicted values of length [n_samples]
- **y_true** (*pd.Series*) – Actual class labels of length [n_samples]
- **X** (*pd.DataFrame or np.ndarray*) – Extra data of shape [n_samples, n_features] necessary to calculate score
- **sample_weight** (*pd.DataFrame or np.ndarray*) – Sample weights used in computing objective value result

**Returns** score

**validate_inputs**(*self*, *y_true*, *y_predicted*)

Validates the input based on a few simple checks.

**Parameters**

- **y_predicted** (*pd.Series, or pd.DataFrame*) – Predicted values of length [n_samples].
- **y_true** (*pd.Series*) – Actual class labels of length [n_samples].

**Raises** **ValueError** – If the inputs are malformed.

**class** evalml.objectives.standard_metrics.**F1Weighted**

F1 score for multiclass classification using weighted averaging.

### Example

```
>>> y_true = pd.Series([0, 1, 0, 2, 0, 1, 2, 1, 2, 0, 2])
>>> y_pred = pd.Series([0, 0, 0, 0, 1, 1, 1, 1, 2, 2, 2])
>>> np.testing.assert_almost_equal(F1Weighted().objective_function(y_true, y_pred),␣
↪0.5454545)
```

**Attributes**

| | |
|---|---|
| **expected_range** | [0, 1] |
| **greater_is_better** | True |
| **is_bounded_like** | True |
| **name** | F1 Weighted |
| **perfect_score** | 1.0 |
| **problem_types** | [ProblemTypes.MULTICLASS, ProblemTypes.TIME_SERIES_MULTICLASS] |
| **score_needs_proba** | False |

**Methods**

| | |
|---|---|
| *calculate_percent_difference* | Calculate the percent difference between scores. |
| *is_defined_for_problem_type* | Returns whether or not an objective is defined for a problem type. |
| *objective_function* | Objective function for F1 score for multiclass classification using weighted averaging. |
| *positive_only* | If True, this objective is only valid for positive data. Defaults to False. |
| *score* | Returns a numerical score indicating performance based on the differences between the predicted and actual values. |
| *validate_inputs* | Validates the input based on a few simple checks. |

**classmethod calculate_percent_difference**(*cls*, *score*, *baseline_score*)

Calculate the percent difference between scores.

**Parameters**

- **score** (*float*) – A score. Output of the score method of this objective.
- **baseline_score** (*float*) – A score. Output of the score method of this objective. In practice, this is the score achieved on this objective with a baseline estimator.

**Returns**

**The percent difference between the scores. Note that for objectives that can be interpreted** as percentages, this will be the difference between the reference score and score. For all other objectives, the difference will be normalized by the reference score.

**Return type** float

`classmethod is_defined_for_problem_type`(*cls*, *problem_type*)

Returns whether or not an objective is defined for a problem type.

`objective_function`(*self*, *y_true*, *y_predicted*, *X=None*, *sample_weight=None*)

Objective function for F1 score for multiclass classification using weighted averaging.

`positive_only`(*cls*)

If True, this objective is only valid for positive data. Defaults to False.

`score`(*self*, *y_true*, *y_predicted*, *X=None*, *sample_weight=None*)

Returns a numerical score indicating performance based on the differences between the predicted and actual values.

**Parameters**

- **y_predicted** (*pd.Series*) – Predicted values of length [n_samples]
- **y_true** (*pd.Series*) – Actual class labels of length [n_samples]
- **X** (*pd.DataFrame or np.ndarray*) – Extra data of shape [n_samples, n_features] necessary to calculate score
- **sample_weight** (*pd.DataFrame or np.ndarray*) – Sample weights used in computing objective value result

**Returns** score

`validate_inputs`(*self*, *y_true*, *y_predicted*)

Validates the input based on a few simple checks.

**Parameters**

- **y_predicted** (*pd.Series, or pd.DataFrame*) – Predicted values of length [n_samples].
- **y_true** (*pd.Series*) – Actual class labels of length [n_samples].

**Raises** **ValueError** – If the inputs are malformed.

**class** `evalml.objectives.standard_metrics.`**Gini**

Gini coefficient for binary classification.

### Example

```
>>> y_true = pd.Series([0, 1, 1, 0, 0, 1, 1, 1, 0, 1, 1])
>>> y_pred = pd.Series([0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1])
>>> np.testing.assert_almost_equal(Gini().objective_function(y_true, y_pred), 0.
↪1428571)
```

**Attributes**

| | |
|---|---|
| **expected_range** | None |
| **greater_is_better** | True |
| **is_bounded_like_percentage** | False |
| **name** | Gini |
| **perfect_score** | 1.0 |
| **problem_types** | [ProblemTypes.BINARY, ProblemTypes.TIME_SERIES_BINARY] |
| **score_needs_proba** | True |

**Methods**

| | |
|---|---|
| *calculate_percent_difference* | Calculate the percent difference between scores. |
| *can_optimize_threshold* | Returns a boolean determining if we can optimize the binary classification objective threshold. |
| *decision_function* | Apply a learned threshold to predicted probabilities to get predicted classes. |
| *is_defined_for_problem_type* | Returns whether or not an objective is defined for a problem type. |
| *objective_function* | Objective function for Gini coefficient for binary classification. |
| *optimize_threshold* | Learn a binary classification threshold which optimizes the current objective. |
| *positive_only* | If True, this objective is only valid for positive data. Defaults to False. |
| *score* | Returns a numerical score indicating performance based on the differences between the predicted and actual values. |
| *validate_inputs* | Validate inputs for scoring. |

**classmethod calculate_percent_difference**(*cls*, *score*, *baseline_score*)

Calculate the percent difference between scores.

**Parameters**

- **score** (*float*) – A score. Output of the score method of this objective.
- **baseline_score** (*float*) – A score. Output of the score method of this objective. In practice, this is the score achieved on this objective with a baseline estimator.

**Returns**

**The percent difference between the scores. Note that for objectives that can be interpreted** as percentages, this will be the difference between the reference score and score. For all other objectives, the difference will be normalized by the reference score.

**Return type** float

**property can_optimize_threshold**(*cls*)

Returns a boolean determining if we can optimize the binary classification objective threshold.

This will be false for any objective that works directly with predicted probabilities, like log loss and AUC. Otherwise, it will be true.

**Returns** Whether or not an objective can be optimized.

**Return type** bool

**decision_function**(*self*, *ypred_proba*, *threshold=0.5*, *X=None*)

Apply a learned threshold to predicted probabilities to get predicted classes.

**Parameters**

- **ypred_proba** (*pd.Series, np.ndarray*) – The classifier's predicted probabilities
- **threshold** (*float, optional*) – Threshold used to make a prediction. Defaults to 0.5.
- **X** (*pd.DataFrame, optional*) – Any extra columns that are needed from training data.

**Returns** predictions

**classmethod is_defined_for_problem_type**(*cls*, *problem_type*)

Returns whether or not an objective is defined for a problem type.

**objective_function**(*self*, *y_true*, *y_predicted*, *X=None*, *sample_weight=None*)

Objective function for Gini coefficient for binary classification.

**optimize_threshold**(*self*, *ypred_proba*, *y_true*, *X=None*)

Learn a binary classification threshold which optimizes the current objective.

**Parameters**

- **ypred_proba** (*pd.Series*) – The classifier's predicted probabilities
- **y_true** (*pd.Series*) – The ground truth for the predictions.
- **X** (*pd.DataFrame, optional*) – Any extra columns that are needed from training data.

**Returns** Optimal threshold for this objective.

**Raises** **RuntimeError** – If objective cannot be optimized.

**positive_only**(*cls*)

If True, this objective is only valid for positive data. Defaults to False.

**score**(*self*, *y_true*, *y_predicted*, *X=None*, *sample_weight=None*)

Returns a numerical score indicating performance based on the differences between the predicted and actual values.

**Parameters**

- **y_predicted** (*pd.Series*) – Predicted values of length [n_samples]
- **y_true** (*pd.Series*) – Actual class labels of length [n_samples]
- **X** (*pd.DataFrame or np.ndarray*) – Extra data of shape [n_samples, n_features] necessary to calculate score
- **sample_weight** (*pd.DataFrame or np.ndarray*) – Sample weights used in computing objective value result

**Returns** score

**validate_inputs**(*self*, *y_true*, *y_predicted*)

Validate inputs for scoring.

**class** evalml.objectives.standard_metrics.**LogLossBinary**

Log Loss for binary classification.

### Example

```
>>> y_true = pd.Series([0, 1, 1, 0, 0, 1, 1, 1, 0, 1, 1])
>>> y_pred = pd.Series([0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1])
>>> np.testing.assert_almost_equal(LogLossBinary().objective_function(y_true, y_
↪pred), 19.6601745)
```

**Attributes**

| | |
|---|---|
| **expected_range** | [0, 1] |
| **greater_is_better** | False |
| **is_bounded_like_a_percentage** | False |
| **name** | Log Loss Binary |
| **perfect_score** | 0.0 |
| **problem_types** | [ProblemTypes.BINARY, ProblemTypes.TIME_SERIES_BINARY] |
| **score_needs_proba** | True |

**Methods**

| | |
|---|---|
| *calculate_percent_difference* | Calculate the percent difference between scores. |
| *can_optimize_threshold* | Returns a boolean determining if we can optimize the binary classification objective threshold. |
| *decision_function* | Apply a learned threshold to predicted probabilities to get predicted classes. |
| *is_defined_for_problem_type* | Returns whether or not an objective is defined for a problem type. |
| *objective_function* | Objective function for log loss for binary classification. |
| *optimize_threshold* | Learn a binary classification threshold which optimizes the current objective. |
| *positive_only* | If True, this objective is only valid for positive data. Defaults to False. |
| *score* | Returns a numerical score indicating performance based on the differences between the predicted and actual values. |
| *validate_inputs* | Validate inputs for scoring. |

**classmethod calculate_percent_difference**(*cls*, *score*, *baseline_score*)

Calculate the percent difference between scores.

**Parameters**

- **score** (*float*) – A score. Output of the score method of this objective.
- **baseline_score** (*float*) – A score. Output of the score method of this objective. In practice, this is the score achieved on this objective with a baseline estimator.

**Returns**

**The percent difference between the scores. Note that for objectives that can be interpreted** as percentages, this will be the difference between the reference score and score. For all other objectives, the difference will be normalized by the reference score.

**Return type** float

**property can_optimize_threshold**(*cls*)

Returns a boolean determining if we can optimize the binary classification objective threshold.

This will be false for any objective that works directly with predicted probabilities, like log loss and AUC. Otherwise, it will be true.

**Returns** Whether or not an objective can be optimized.

**Return type** bool

**decision_function**(*self*, *ypred_proba*, *threshold=0.5*, *X=None*)

Apply a learned threshold to predicted probabilities to get predicted classes.

**Parameters**

- **ypred_proba** (*pd.Series, np.ndarray*) – The classifier's predicted probabilities
- **threshold** (*float, optional*) – Threshold used to make a prediction. Defaults to 0.5.
- **X** (*pd.DataFrame, optional*) – Any extra columns that are needed from training data.

**Returns** predictions

**classmethod is_defined_for_problem_type**(*cls*, *problem_type*)

Returns whether or not an objective is defined for a problem type.

**objective_function**(*self*, *y_true*, *y_predicted*, *X=None*, *sample_weight=None*)

Objective function for log loss for binary classification.

**optimize_threshold**(*self*, *ypred_proba*, *y_true*, *X=None*)

Learn a binary classification threshold which optimizes the current objective.

**Parameters**

- **ypred_proba** (*pd.Series*) – The classifier's predicted probabilities
- **y_true** (*pd.Series*) – The ground truth for the predictions.
- **X** (*pd.DataFrame, optional*) – Any extra columns that are needed from training data.

**Returns** Optimal threshold for this objective.

**Raises** **RuntimeError** – If objective cannot be optimized.

**positive_only**(*cls*)

If True, this objective is only valid for positive data. Defaults to False.

**score**(*self*, *y_true*, *y_predicted*, *X=None*, *sample_weight=None*)

Returns a numerical score indicating performance based on the differences between the predicted and actual values.

**Parameters**

- **y_predicted** (*pd.Series*) – Predicted values of length [n_samples]
- **y_true** (*pd.Series*) – Actual class labels of length [n_samples]
- **X** (*pd.DataFrame or np.ndarray*) – Extra data of shape [n_samples, n_features] necessary to calculate score
- **sample_weight** (*pd.DataFrame or np.ndarray*) – Sample weights used in computing objective value result

**Returns** score

**validate_inputs**(*self*, *y_true*, *y_predicted*)

Validate inputs for scoring.

**class** evalml.objectives.standard_metrics.**LogLossMulticlass**

Log Loss for multiclass classification.

### Example

```
>>> y_true = [0, 1, 2, 0, 2, 1]
>>> y_pred = [[0.7, 0.2, 0.1],
...           [0.3, 0.5, 0.2],
...           [0.1, 0.3, 0.6],
...           [0.9, 0.1, 0.0],
...           [0.3, 0.1, 0.6],
...           [0.5, 0.5, 0.0]]
>>> np.testing.assert_almost_equal(LogLossMulticlass().objective_function(y_true, y_
↪pred), 0.4783301)
```

**Attributes**

| | |
|---|---|
| **expected_range** | [0, 1] |
| **greater_is_better** | False |
| **is_bounded_like_a_percentage** | False |
| **name** | Log Loss Multiclass |
| **perfect_score** | 0.0 |
| **problem_types** | [ProblemTypes.MULTICLASS, ProblemTypes.TIME_SERIES_MULTICLASS] |
| **score_needs_proba** | True |

**Methods**

| | |
|---|---|
| *calculate_percent_difference* | Calculate the percent difference between scores. |
| *is_defined_for_problem_type* | Returns whether or not an objective is defined for a problem type. |
| *objective_function* | Objective function for log loss for multiclass classification. |
| *positive_only* | If True, this objective is only valid for positive data. Defaults to False. |
| *score* | Returns a numerical score indicating performance based on the differences between the predicted and actual values. |
| *validate_inputs* | Validates the input based on a few simple checks. |

**classmethod calculate_percent_difference**(*cls*, *score*, *baseline_score*)

Calculate the percent difference between scores.

**Parameters**

- **score** (*float*) – A score. Output of the score method of this objective.

- **baseline_score** (*float*) – A score. Output of the score method of this objective. In practice, this is the score achieved on this objective with a baseline estimator.

**Returns**

**The percent difference between the scores. Note that for objectives that can be interpreted** as percentages, this will be the difference between the reference score and score. For all other objectives, the difference will be normalized by the reference score.

**Return type** float

**classmethod is_defined_for_problem_type**(*cls*, *problem_type*)

Returns whether or not an objective is defined for a problem type.

**objective_function**(*self*, *y_true*, *y_predicted*, *X=None*, *sample_weight=None*)

Objective function for log loss for multiclass classification.

**positive_only**(*cls*)

If True, this objective is only valid for positive data. Defaults to False.

**score**(*self*, *y_true*, *y_predicted*, *X=None*, *sample_weight=None*)

Returns a numerical score indicating performance based on the differences between the predicted and actual values.

**Parameters**

- **y_predicted** (*pd.Series*) – Predicted values of length [n_samples]
- **y_true** (*pd.Series*) – Actual class labels of length [n_samples]
- **X** (*pd.DataFrame or np.ndarray*) – Extra data of shape [n_samples, n_features] necessary to calculate score
- **sample_weight** (*pd.DataFrame or np.ndarray*) – Sample weights used in computing objective value result

**Returns** score

**validate_inputs**(*self*, *y_true*, *y_predicted*)

Validates the input based on a few simple checks.

**Parameters**

- **y_predicted** (*pd.Series, or pd.DataFrame*) – Predicted values of length [n_samples].
- **y_true** (*pd.Series*) – Actual class labels of length [n_samples].

**Raises** **ValueError** – If the inputs are malformed.

**class** evalml.objectives.standard_metrics.**MAE**

Mean absolute error for regression.

### Example

```
>>> y_true = pd.Series([1.5, 2, 3, 1, 0.5, 1, 2.5, 2.5, 1, 0.5, 2])
>>> y_pred = pd.Series([1.5, 2.5, 2, 1, 0.5, 1, 3, 2.25, 0.75, 0.25, 1.75])
>>> np.testing.assert_almost_equal(MAE().objective_function(y_true, y_pred), 0.
↪2727272)
```

**Attributes**

| | |
|---|---|
| **expected_range** | None |
| **greater_is_better** | False |
| **is_bounded_like** | True |
| **name** | MAE |
| **perfect_score** | 0.0 |
| **problem_types** | [ProblemTypes.REGRESSION, ProblemTypes.TIME_SERIES_REGRESSION] |
| **score_needs_proba** | False |

**Methods**

| | |
|---|---|
| *calculate_percent_difference* | Calculate the percent difference between scores. |
| *is_defined_for_problem_type* | Returns whether or not an objective is defined for a problem type. |
| *objective_function* | Objective function for mean absolute error for regression. |
| *positive_only* | If True, this objective is only valid for positive data. Defaults to False. |
| *score* | Returns a numerical score indicating performance based on the differences between the predicted and actual values. |
| *validate_inputs* | Validates the input based on a few simple checks. |

**classmethod calculate_percent_difference**(*cls*, *score*, *baseline_score*)

Calculate the percent difference between scores.

**Parameters**

- **score** (*float*) – A score. Output of the score method of this objective.
- **baseline_score** (*float*) – A score. Output of the score method of this objective. In practice, this is the score achieved on this objective with a baseline estimator.

**Returns**

**The percent difference between the scores. Note that for objectives that can be interpreted** as percentages, this will be the difference between the reference score and score. For all other objectives, the difference will be normalized by the reference score.

**Return type** float

**classmethod is_defined_for_problem_type**(*cls*, *problem_type*)

Returns whether or not an objective is defined for a problem type.

**objective_function**(*self*, *y_true*, *y_predicted*, *X=None*, *sample_weight=None*)

Objective function for mean absolute error for regression.

**positive_only**(*cls*)

If True, this objective is only valid for positive data. Defaults to False.

**score**(*self*, *y_true*, *y_predicted*, *X=None*, *sample_weight=None*)

Returns a numerical score indicating performance based on the differences between the predicted and actual values.

**Parameters**

- **y_predicted** (*pd.Series*) – Predicted values of length [n_samples]
- **y_true** (*pd.Series*) – Actual class labels of length [n_samples]
- **X** (*pd.DataFrame or np.ndarray*) – Extra data of shape [n_samples, n_features] necessary to calculate score
- **sample_weight** (*pd.DataFrame or np.ndarray*) – Sample weights used in computing objective value result

**Returns** score

**validate_inputs**(*self*, *y_true*, *y_predicted*)

Validates the input based on a few simple checks.

**Parameters**

- **y_predicted** (*pd.Series, or pd.DataFrame*) – Predicted values of length [n_samples].
- **y_true** (*pd.Series*) – Actual class labels of length [n_samples].

**Raises** **ValueError** – If the inputs are malformed.

**class** evalml.objectives.standard_metrics.**MAPE**

Mean absolute percentage error for time series regression. Scaled by 100 to return a percentage.

Only valid for nonzero inputs. Otherwise, will throw a ValueError.

### Example

```
>>> y_true = pd.Series([1.5, 2, 3, 1, 0.5, 1, 2.5, 2.5, 1, 0.5, 2])
>>> y_pred = pd.Series([1.5, 2.5, 2, 1, 0.5, 1, 3, 2.25, 0.75, 0.25, 1.75])
>>> np.testing.assert_almost_equal(MAPE().objective_function(y_true, y_pred), 15.
↪9848484)
```

**Attributes**

| | |
|---|---|
| **expected_range** | None |
| **greater_is_better** | False |
| **is_bounded_like_percentage** | False |
| **name** | Mean Absolute Percentage Error |
| **perfect_score** | 0.0 |
| **problem_types** | [ProblemTypes.TIME_SERIES_REGRESSION] |
| **score_needs_proba** | False |

**Methods**

| | |
|---|---|
| *calculate_percent_difference* | Calculate the percent difference between scores. |
| *is_defined_for_problem_type* | Returns whether or not an objective is defined for a problem type. |
| *objective_function* | Objective function for mean absolute percentage error for time series regression. |
| *positive_only* | If True, this objective is only valid for positive data. |
| *score* | Returns a numerical score indicating performance based on the differences between the predicted and actual values. |
| *validate_inputs* | Validates the input based on a few simple checks. |

**classmethod calculate_percent_difference**(*cls*, *score*, *baseline_score*)

Calculate the percent difference between scores.

**Parameters**

- **score** (*float*) – A score. Output of the score method of this objective.
- **baseline_score** (*float*) – A score. Output of the score method of this objective. In practice, this is the score achieved on this objective with a baseline estimator.

**Returns**

**The percent difference between the scores. Note that for objectives that can be interpreted** as percentages, this will be the difference between the reference score and score. For all other objectives, the difference will be normalized by the reference score.

**Return type** float

**classmethod is_defined_for_problem_type**(*cls*, *problem_type*)

Returns whether or not an objective is defined for a problem type.

**objective_function**(*self*, *y_true*, *y_predicted*, *X=None*, *sample_weight=None*)

Objective function for mean absolute percentage error for time series regression.

**positive_only**(*self*)

If True, this objective is only valid for positive data.

**score**(*self*, *y_true*, *y_predicted*, *X=None*, *sample_weight=None*)

Returns a numerical score indicating performance based on the differences between the predicted and actual values.

**Parameters**

- **y_predicted** (*pd.Series*) – Predicted values of length [n_samples]
- **y_true** (*pd.Series*) – Actual class labels of length [n_samples]
- **X** (*pd.DataFrame or np.ndarray*) – Extra data of shape [n_samples, n_features] necessary to calculate score
- **sample_weight** (*pd.DataFrame or np.ndarray*) – Sample weights used in computing objective value result

**Returns** score

**validate_inputs**(*self*, *y_true*, *y_predicted*)

Validates the input based on a few simple checks.

**Parameters**

- **y_predicted** (*pd.Series, or pd.DataFrame*) – Predicted values of length [n_samples].
- **y_true** (*pd.Series*) – Actual class labels of length [n_samples].

**Raises** **ValueError** – If the inputs are malformed.

**class** evalml.objectives.standard_metrics.**MaxError**

Maximum residual error for regression.

### Example

```
>>> y_true = pd.Series([1.5, 2, 3, 1, 0.5, 1, 2.5, 2.5, 1, 0.5, 2])
>>> y_pred = pd.Series([1.5, 2.5, 2, 1, 0.5, 1, 3, 2.25, 0.75, 0.25, 1.75])
>>> np.testing.assert_almost_equal(MaxError().objective_function(y_true, y_pred), 1.
↪0)
```

**Attributes**

| | |
|---|---|
| **expected_range** | None |
| **greater_is_better** | False |
| **is_bounded_like_percentage** | False |
| **name** | MaxError |
| **perfect_score** | 0.0 |
| **problem_types** | [ProblemTypes.REGRESSION, ProblemTypes.TIME_SERIES_REGRESSION] |
| **score_needs_proba** | False |

**Methods**

| | |
|---|---|
| *calculate_percent_difference* | Calculate the percent difference between scores. |
| *is_defined_for_problem_type* | Returns whether or not an objective is defined for a problem type. |
| *objective_function* | Objective function for maximum residual error for regression. |
| *positive_only* | If True, this objective is only valid for positive data. Defaults to False. |
| *score* | Returns a numerical score indicating performance based on the differences between the predicted and actual values. |
| *validate_inputs* | Validates the input based on a few simple checks. |

**classmethod calculate_percent_difference**(*cls*, *score*, *baseline_score*)

Calculate the percent difference between scores.

**Parameters**

- **score** (*float*) – A score. Output of the score method of this objective.
- **baseline_score** (*float*) – A score. Output of the score method of this objective. In practice, this is the score achieved on this objective with a baseline estimator.

**Returns**

**The percent difference between the scores. Note that for objectives that can be interpreted** as percentages, this will be the difference between the reference score and score. For all other objectives, the difference will be normalized by the reference score.

**Return type** float

**classmethod is_defined_for_problem_type**(*cls*, *problem_type*)

Returns whether or not an objective is defined for a problem type.

**objective_function**(*self*, *y_true*, *y_predicted*, *X=None*, *sample_weight=None*)

Objective function for maximum residual error for regression.

**positive_only**(*cls*)

If True, this objective is only valid for positive data. Defaults to False.

**score**(*self*, *y_true*, *y_predicted*, *X=None*, *sample_weight=None*)

Returns a numerical score indicating performance based on the differences between the predicted and actual values.

**Parameters**

- **y_predicted** (*pd.Series*) – Predicted values of length [n_samples]
- **y_true** (*pd.Series*) – Actual class labels of length [n_samples]
- **X** (*pd.DataFrame or np.ndarray*) – Extra data of shape [n_samples, n_features] necessary to calculate score
- **sample_weight** (*pd.DataFrame or np.ndarray*) – Sample weights used in computing objective value result

**Returns** score

**validate_inputs**(*self*, *y_true*, *y_predicted*)

Validates the input based on a few simple checks.

**Parameters**

- **y_predicted** (*pd.Series, or pd.DataFrame*) – Predicted values of length [n_samples].
- **y_true** (*pd.Series*) – Actual class labels of length [n_samples].

**Raises** **ValueError** – If the inputs are malformed.

**class** evalml.objectives.standard_metrics.**MCCBinary**

Matthews correlation coefficient for binary classification.

### Example

```
>>> y_true = pd.Series([0, 1, 1, 0, 0, 1, 1, 1, 0, 1, 1])
>>> y_pred = pd.Series([0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1])
>>> np.testing.assert_almost_equal(MCCBinary().objective_function(y_true, y_pred),␣
↪0.2390457)
```

**Attributes**

| | |
|---|---|
| **expected_range** | None |
| **greater_is_better** | True |
| **is_bounded_like_percentage** | False |
| **name** | MCC Binary |
| **perfect_score** | 1.0 |
| **problem_types** | [ProblemTypes.BINARY, ProblemTypes.TIME_SERIES_BINARY] |
| **score_needs_proba** | False |

**Methods**

| | |
|---|---|
| *calculate_percent_difference* | Calculate the percent difference between scores. |
| *can_optimize_threshold* | Returns a boolean determining if we can optimize the binary classification objective threshold. |
| *decision_function* | Apply a learned threshold to predicted probabilities to get predicted classes. |
| *is_defined_for_problem_type* | Returns whether or not an objective is defined for a problem type. |
| *objective_function* | Objective function for Matthews correlation coefficient for binary classification. |
| *optimize_threshold* | Learn a binary classification threshold which optimizes the current objective. |
| *positive_only* | If True, this objective is only valid for positive data. Defaults to False. |
| *score* | Returns a numerical score indicating performance based on the differences between the predicted and actual values. |
| *validate_inputs* | Validate inputs for scoring. |

**classmethod calculate_percent_difference**(*cls*, *score*, *baseline_score*)

Calculate the percent difference between scores.

**Parameters**

- **score** (*float*) – A score. Output of the score method of this objective.
- **baseline_score** (*float*) – A score. Output of the score method of this objective. In practice, this is the score achieved on this objective with a baseline estimator.

**Returns**

**The percent difference between the scores. Note that for objectives that can be interpreted** as percentages, this will be the difference between the reference score and score. For all other objectives, the difference will be normalized by the reference score.

**Return type** float

**property can_optimize_threshold**(*cls*)

Returns a boolean determining if we can optimize the binary classification objective threshold.

This will be false for any objective that works directly with predicted probabilities, like log loss and AUC. Otherwise, it will be true.

**Returns** Whether or not an objective can be optimized.

**Return type** bool

**decision_function**(*self*, *ypred_proba*, *threshold=0.5*, *X=None*)

Apply a learned threshold to predicted probabilities to get predicted classes.

**Parameters**

- **ypred_proba** (*pd.Series, np.ndarray*) – The classifier's predicted probabilities
- **threshold** (*float, optional*) – Threshold used to make a prediction. Defaults to 0.5.
- **X** (*pd.DataFrame, optional*) – Any extra columns that are needed from training data.

**Returns** predictions

**classmethod is_defined_for_problem_type**(*cls*, *problem_type*)

Returns whether or not an objective is defined for a problem type.

**objective_function**(*self*, *y_true*, *y_predicted*, *X=None*, *sample_weight=None*)

Objective function for Matthews correlation coefficient for binary classification.

**optimize_threshold**(*self*, *ypred_proba*, *y_true*, *X=None*)

Learn a binary classification threshold which optimizes the current objective.

**Parameters**

- **ypred_proba** (*pd.Series*) – The classifier's predicted probabilities
- **y_true** (*pd.Series*) – The ground truth for the predictions.
- **X** (*pd.DataFrame, optional*) – Any extra columns that are needed from training data.

**Returns** Optimal threshold for this objective.

**Raises** **RuntimeError** – If objective cannot be optimized.

**positive_only**(*cls*)

If True, this objective is only valid for positive data. Defaults to False.

**score**(*self*, *y_true*, *y_predicted*, *X=None*, *sample_weight=None*)

Returns a numerical score indicating performance based on the differences between the predicted and actual values.

**Parameters**

- **y_predicted** (*pd.Series*) – Predicted values of length [n_samples]
- **y_true** (*pd.Series*) – Actual class labels of length [n_samples]
- **X** (*pd.DataFrame or np.ndarray*) – Extra data of shape [n_samples, n_features] necessary to calculate score
- **sample_weight** (*pd.DataFrame or np.ndarray*) – Sample weights used in computing objective value result

**Returns** score

**validate_inputs**(*self*, *y_true*, *y_predicted*)

Validate inputs for scoring.

**class** evalml.objectives.standard_metrics.**MCCMulticlass**

Matthews correlation coefficient for multiclass classification.

### Example

```
>>> y_true = pd.Series([0, 1, 0, 2, 0, 1, 2, 1, 2, 0, 2])
>>> y_pred = pd.Series([0, 0, 0, 0, 1, 1, 1, 1, 2, 2, 2])
>>> np.testing.assert_almost_equal(MCCMulticlass().objective_function(y_true, y_
↪pred), 0.325)
```

**Attributes**

| | |
|---|---|
| **expected_range** | None |
| **greater_is_better** | True |
| **is_bounded_like_percentage** | False |
| **name** | MCC Multiclass |
| **perfect_score** | 1.0 |
| **problem_types** | [ProblemTypes.MULTICLASS, ProblemTypes.TIME_SERIES_MULTICLASS] |
| **score_needs_proba** | False |

**Methods**

| | |
|---|---|
| *calculate_percent_difference* | Calculate the percent difference between scores. |
| *is_defined_for_problem_type* | Returns whether or not an objective is defined for a problem type. |
| *objective_function* | Objective function for Matthews correlation coefficient for multiclass classification. |
| *positive_only* | If True, this objective is only valid for positive data. Defaults to False. |
| *score* | Returns a numerical score indicating performance based on the differences between the predicted and actual values. |
| *validate_inputs* | Validates the input based on a few simple checks. |

**classmethod calculate_percent_difference**(*cls*, *score*, *baseline_score*)

Calculate the percent difference between scores.

**Parameters**

- **score** (*float*) – A score. Output of the score method of this objective.
- **baseline_score** (*float*) – A score. Output of the score method of this objective. In practice, this is the score achieved on this objective with a baseline estimator.

**Returns**

**The percent difference between the scores. Note that for objectives that can be interpreted** as percentages, this will be the difference between the reference score and score. For all other objectives, the difference will be normalized by the reference score.

**Return type** float

**classmethod is_defined_for_problem_type**(*cls*, *problem_type*)

Returns whether or not an objective is defined for a problem type.

**objective_function**(*self*, *y_true*, *y_predicted*, *X=None*, *sample_weight=None*)

Objective function for Matthews correlation coefficient for multiclass classification.

**positive_only**(*cls*)

If True, this objective is only valid for positive data. Defaults to False.

**score**(*self*, *y_true*, *y_predicted*, *X=None*, *sample_weight=None*)

Returns a numerical score indicating performance based on the differences between the predicted and actual values.

**Parameters**

- **y_predicted** (*pd.Series*) – Predicted values of length [n_samples]
- **y_true** (*pd.Series*) – Actual class labels of length [n_samples]
- **X** (*pd.DataFrame or np.ndarray*) – Extra data of shape [n_samples, n_features] necessary to calculate score
- **sample_weight** (*pd.DataFrame or np.ndarray*) – Sample weights used in computing objective value result

**Returns** score

**validate_inputs**(*self*, *y_true*, *y_predicted*)

Validates the input based on a few simple checks.

**Parameters**

- **y_predicted** (*pd.Series, or pd.DataFrame*) – Predicted values of length [n_samples].
- **y_true** (*pd.Series*) – Actual class labels of length [n_samples].

**Raises** **ValueError** – If the inputs are malformed.

**class** evalml.objectives.standard_metrics.**MeanSquaredLogError**

Mean squared log error for regression.

Only valid for nonnegative inputs. Otherwise, will throw a ValueError.

### Example

```
>>> y_true = pd.Series([1.5, 2, 3, 1, 0.5, 1, 2.5, 2.5, 1, 0.5, 2])
>>> y_pred = pd.Series([1.5, 2.5, 2, 1, 0.5, 1, 3, 2.25, 0.75, 0.25, 1.75])
>>> np.testing.assert_almost_equal(MeanSquaredLogError().objective_function(y_true,␣
↪y_pred), 0.0171353)
```

**Attributes**

| | |
|---|---|
| **expected_range** | None |
| **greater_is_better** | False |
| **is_bounded_like_percentage** | False |
| **name** | Mean Squared Log Error |
| **perfect_score** | 0.0 |
| **problem_types** | [ProblemTypes.REGRESSION, ProblemTypes.TIME_SERIES_REGRESSION] |
| **score_needs_proba** | False |

**Methods**

| | |
|---|---|
| *calculate_percent_difference* | Calculate the percent difference between scores. |
| *is_defined_for_problem_type* | Returns whether or not an objective is defined for a problem type. |
| *objective_function* | Objective function for mean squared log error for regression. |
| *positive_only* | If True, this objective is only valid for positive data. |
| *score* | Returns a numerical score indicating performance based on the differences between the predicted and actual values. |
| *validate_inputs* | Validates the input based on a few simple checks. |

**classmethod calculate_percent_difference**(*cls*, *score*, *baseline_score*)

Calculate the percent difference between scores.

**Parameters**

- **score** (*float*) – A score. Output of the score method of this objective.

- **baseline_score** (*float*) – A score. Output of the score method of this objective. In practice, this is the score achieved on this objective with a baseline estimator.

**Returns**

**The percent difference between the scores. Note that for objectives that can be interpreted** as percentages, this will be the difference between the reference score and score. For all other objectives, the difference will be normalized by the reference score.

**Return type** float

**classmethod is_defined_for_problem_type**(*cls*, *problem_type*)

Returns whether or not an objective is defined for a problem type.

**objective_function**(*self*, *y_true*, *y_predicted*, *X=None*, *sample_weight=None*)

Objective function for mean squared log error for regression.

**positive_only**(*self*)

If True, this objective is only valid for positive data.

**score**(*self*, *y_true*, *y_predicted*, *X=None*, *sample_weight=None*)

Returns a numerical score indicating performance based on the differences between the predicted and actual values.

**Parameters**

- **y_predicted** (*pd.Series*) – Predicted values of length [n_samples]
- **y_true** (*pd.Series*) – Actual class labels of length [n_samples]
- **X** (*pd.DataFrame or np.ndarray*) – Extra data of shape [n_samples, n_features] necessary to calculate score
- **sample_weight** (*pd.DataFrame or np.ndarray*) – Sample weights used in computing objective value result

**Returns** score

**validate_inputs**(*self*, *y_true*, *y_predicted*)

Validates the input based on a few simple checks.

**Parameters**

- **y_predicted** (*pd.Series, or pd.DataFrame*) – Predicted values of length [n_samples].
- **y_true** (*pd.Series*) – Actual class labels of length [n_samples].

**Raises** **ValueError** – If the inputs are malformed.

**class** evalml.objectives.standard_metrics.**MedianAE**

Median absolute error for regression.

### Example

```
>>> y_true = pd.Series([1.5, 2, 3, 1, 0.5, 1, 2.5, 2.5, 1, 0.5, 2])
>>> y_pred = pd.Series([1.5, 2.5, 2, 1, 0.5, 1, 3, 2.25, 0.75, 0.25, 1.75])
>>> np.testing.assert_almost_equal(MedianAE().objective_function(y_true, y_pred), 0.
↪25)
```

**Attributes**

| | |
|---|---|
| **expected_range** | None |
| **greater_is_better** | False |
| **is_bounded_like_percentage** | False |
| **name** | MedianAE |
| **perfect_score** | 0.0 |
| **problem_types** | [ProblemTypes.REGRESSION, ProblemTypes.TIME_SERIES_REGRESSION] |
| **score_needs_proba** | False |

**Methods**

| | |
|---|---|
| *calculate_percent_difference* | Calculate the percent difference between scores. |
| *is_defined_for_problem_type* | Returns whether or not an objective is defined for a problem type. |
| *objective_function* | Objective function for median absolute error for regression. |
| *positive_only* | If True, this objective is only valid for positive data. Defaults to False. |
| *score* | Returns a numerical score indicating performance based on the differences between the predicted and actual values. |
| *validate_inputs* | Validates the input based on a few simple checks. |

**classmethod calculate_percent_difference**(*cls*, *score*, *baseline_score*)

Calculate the percent difference between scores.

**Parameters**

- **score** (*float*) – A score. Output of the score method of this objective.
- **baseline_score** (*float*) – A score. Output of the score method of this objective. In practice, this is the score achieved on this objective with a baseline estimator.

**Returns**

**The percent difference between the scores. Note that for objectives that can be interpreted** as percentages, this will be the difference between the reference score and score. For all other objectives, the difference will be normalized by the reference score.

**Return type** float

**classmethod is_defined_for_problem_type**(*cls*, *problem_type*)

Returns whether or not an objective is defined for a problem type.

**objective_function**(*self*, *y_true*, *y_predicted*, *X=None*, *sample_weight=None*)

Objective function for median absolute error for regression.

**positive_only**(*cls*)

If True, this objective is only valid for positive data. Defaults to False.

**score**(*self*, *y_true*, *y_predicted*, *X=None*, *sample_weight=None*)

Returns a numerical score indicating performance based on the differences between the predicted and actual values.

**Parameters**

- **y_predicted** (*pd.Series*) – Predicted values of length [n_samples]
- **y_true** (*pd.Series*) – Actual class labels of length [n_samples]
- **X** (*pd.DataFrame or np.ndarray*) – Extra data of shape [n_samples, n_features] necessary to calculate score
- **sample_weight** (*pd.DataFrame or np.ndarray*) – Sample weights used in computing objective value result

**Returns** score

**validate_inputs**(*self*, *y_true*, *y_predicted*)

Validates the input based on a few simple checks.

**Parameters**

- **y_predicted** (*pd.Series, or pd.DataFrame*) – Predicted values of length [n_samples].
- **y_true** (*pd.Series*) – Actual class labels of length [n_samples].

**Raises** **ValueError** – If the inputs are malformed.

**class** evalml.objectives.standard_metrics.**MSE**

Mean squared error for regression.

### Example

```
>>> y_true = pd.Series([1.5, 2, 3, 1, 0.5, 1, 2.5, 2.5, 1, 0.5, 2])
>>> y_pred = pd.Series([1.5, 2.5, 2, 1, 0.5, 1, 3, 2.25, 0.75, 0.25, 1.75])
>>> np.testing.assert_almost_equal(MSE().objective_function(y_true, y_pred), 0.
↪1590909)
```

**Attributes**

| | |
|---|---|
| **expected_range** | None |
| **greater_is_better** | False |
| **is_bounded_like_percentage** | False |
| **name** | MSE |
| **perfect_score** | 0.0 |
| **problem_types** | [ProblemTypes.REGRESSION, ProblemTypes.TIME_SERIES_REGRESSION] |
| **score_needs_proba** | False |

**Methods**

| | |
|---|---|
| *calculate_percent_difference* | Calculate the percent difference between scores. |
| *is_defined_for_problem_type* | Returns whether or not an objective is defined for a problem type. |
| *objective_function* | Objective function for mean squared error for regression. |
| *positive_only* | If True, this objective is only valid for positive data. Defaults to False. |
| *score* | Returns a numerical score indicating performance based on the differences between the predicted and actual values. |
| *validate_inputs* | Validates the input based on a few simple checks. |

**classmethod calculate_percent_difference**(*cls*, *score*, *baseline_score*)

Calculate the percent difference between scores.

**Parameters**

- **score** (*float*) – A score. Output of the score method of this objective.
- **baseline_score** (*float*) – A score. Output of the score method of this objective. In practice, this is the score achieved on this objective with a baseline estimator.

**Returns**

**The percent difference between the scores. Note that for objectives that can be interpreted** as percentages, this will be the difference between the reference score and score. For all other objectives, the difference will be normalized by the reference score.

**Return type** float

**classmethod is_defined_for_problem_type**(*cls*, *problem_type*)

Returns whether or not an objective is defined for a problem type.

**objective_function**(*self*, *y_true*, *y_predicted*, *X=None*, *sample_weight=None*)

Objective function for mean squared error for regression.

**positive_only**(*cls*)

If True, this objective is only valid for positive data. Defaults to False.

**score**(*self*, *y_true*, *y_predicted*, *X=None*, *sample_weight=None*)

Returns a numerical score indicating performance based on the differences between the predicted and actual values.

**Parameters**

- **y_predicted** (*pd.Series*) – Predicted values of length [n_samples]
- **y_true** (*pd.Series*) – Actual class labels of length [n_samples]
- **X** (*pd.DataFrame or np.ndarray*) – Extra data of shape [n_samples, n_features] necessary to calculate score
- **sample_weight** (*pd.DataFrame or np.ndarray*) – Sample weights used in computing objective value result

**Returns** score

**validate_inputs**(*self*, *y_true*, *y_predicted*)

Validates the input based on a few simple checks.

**Parameters**

- **y_predicted** (*pd.Series, or pd.DataFrame*) – Predicted values of length [n_samples].
- **y_true** (*pd.Series*) – Actual class labels of length [n_samples].

**Raises** **ValueError** – If the inputs are malformed.

**class** evalml.objectives.standard_metrics.**Precision**

Precision score for binary classification.

### Example

```
>>> y_true = pd.Series([0, 1, 1, 0, 0, 1, 1, 1, 0, 1, 1])
>>> y_pred = pd.Series([0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1])
>>> np.testing.assert_almost_equal(Precision().objective_function(y_true, y_pred),␣
↪1.0)
```

**Attributes**

| | |
|---|---|
| **expected_range** | [0, 1] |
| **greater_is_better** | True |
| **is_bounded_like** | True |
| **name** | Percentage |
| **name** | Precision |
| **perfect_score** | 1.0 |
| **problem_types** | [ProblemTypes.BINARY, ProblemTypes.TIME_SERIES_BINARY] |
| **score_needs_proba** | False |

**Methods**

| | |
|---|---|
| *calculate_percent_difference* | Calculate the percent difference between scores. |
| *can_optimize_threshold* | Returns a boolean determining if we can optimize the binary classification objective threshold. |
| *decision_function* | Apply a learned threshold to predicted probabilities to get predicted classes. |
| *is_defined_for_problem_type* | Returns whether or not an objective is defined for a problem type. |
| *objective_function* | Objective function for precision score for binary classification. |
| *optimize_threshold* | Learn a binary classification threshold which optimizes the current objective. |
| *positive_only* | If True, this objective is only valid for positive data. Defaults to False. |
| *score* | Returns a numerical score indicating performance based on the differences between the predicted and actual values. |
| *validate_inputs* | Validate inputs for scoring. |

**classmethod calculate_percent_difference**(*cls*, *score*, *baseline_score*)

Calculate the percent difference between scores.

**Parameters**

- **score** (*float*) – A score. Output of the score method of this objective.
- **baseline_score** (*float*) – A score. Output of the score method of this objective. In practice, this is the score achieved on this objective with a baseline estimator.

**Returns**

**The percent difference between the scores. Note that for objectives that can be interpreted** as percentages, this will be the difference between the reference score and score. For all other objectives, the difference will be normalized by the reference score.

**Return type** float

**property can_optimize_threshold**(*cls*)

Returns a boolean determining if we can optimize the binary classification objective threshold.

This will be false for any objective that works directly with predicted probabilities, like log loss and AUC. Otherwise, it will be true.

**Returns** Whether or not an objective can be optimized.

**Return type** bool

**decision_function**(*self*, *ypred_proba*, *threshold=0.5*, *X=None*)

Apply a learned threshold to predicted probabilities to get predicted classes.

**Parameters**

- **ypred_proba** (*pd.Series, np.ndarray*) – The classifier's predicted probabilities
- **threshold** (*float, optional*) – Threshold used to make a prediction. Defaults to 0.5.
- **X** (*pd.DataFrame, optional*) – Any extra columns that are needed from training data.

**Returns** predictions

**classmethod is_defined_for_problem_type**(*cls*, *problem_type*)

Returns whether or not an objective is defined for a problem type.

**objective_function**(*self*, *y_true*, *y_predicted*, *X=None*, *sample_weight=None*)

Objective function for precision score for binary classification.

**optimize_threshold**(*self*, *ypred_proba*, *y_true*, *X=None*)

Learn a binary classification threshold which optimizes the current objective.

**Parameters**

- **ypred_proba** (*pd.Series*) – The classifier's predicted probabilities
- **y_true** (*pd.Series*) – The ground truth for the predictions.
- **X** (*pd.DataFrame, optional*) – Any extra columns that are needed from training data.

**Returns** Optimal threshold for this objective.

**Raises** **RuntimeError** – If objective cannot be optimized.

**positive_only**(*cls*)

If True, this objective is only valid for positive data. Defaults to False.

**score**(*self*, *y_true*, *y_predicted*, *X=None*, *sample_weight=None*)

Returns a numerical score indicating performance based on the differences between the predicted and actual values.

**Parameters**

- **y_predicted** (*pd.Series*) – Predicted values of length [n_samples]
- **y_true** (*pd.Series*) – Actual class labels of length [n_samples]
- **X** (*pd.DataFrame or np.ndarray*) – Extra data of shape [n_samples, n_features] necessary to calculate score
- **sample_weight** (*pd.DataFrame or np.ndarray*) – Sample weights used in computing objective value result

**Returns** score

**validate_inputs**(*self*, *y_true*, *y_predicted*)

Validate inputs for scoring.

**class** evalml.objectives.standard_metrics.**PrecisionMacro**

Precision score for multiclass classification using macro-averaging.

### Example

```
>>> y_true = pd.Series([0, 1, 0, 2, 0, 1, 2, 1, 2, 0, 2])
>>> y_pred = pd.Series([0, 0, 0, 0, 1, 1, 1, 1, 2, 2, 2])
>>> np.testing.assert_almost_equal(PrecisionMacro().objective_function(y_true, y_
↪pred), 0.5555555)
```

**Attributes**

| | |
|---|---|
| **expected_range** | [0, 1] |
| **greater_is_better** | True |
| **is_bounded_like** | True |
| **name** | Precision Macro |
| **perfect_score** | 1.0 |
| **problem_types** | [ProblemTypes.MULTICLASS, ProblemTypes.TIME_SERIES_MULTICLASS] |
| **score_needs_proba** | False |

**Methods**

| | |
|---|---|
| *calculate_percent_difference* | Calculate the percent difference between scores. |
| *is_defined_for_problem_type* | Returns whether or not an objective is defined for a problem type. |
| *objective_function* | Objective function for precision score for multiclass classification using macro-averaging. |
| *positive_only* | If True, this objective is only valid for positive data. Defaults to False. |
| *score* | Returns a numerical score indicating performance based on the differences between the predicted and actual values. |
| *validate_inputs* | Validates the input based on a few simple checks. |

**classmethod calculate_percent_difference**(*cls*, *score*, *baseline_score*)

Calculate the percent difference between scores.

**Parameters**

- **score** (*float*) – A score. Output of the score method of this objective.
- **baseline_score** (*float*) – A score. Output of the score method of this objective. In practice, this is the score achieved on this objective with a baseline estimator.

**Returns**

**The percent difference between the scores. Note that for objectives that can be interpreted** as percentages, this will be the difference between the reference score and score. For all other objectives, the difference will be normalized by the reference score.

**Return type** float

**classmethod is_defined_for_problem_type**(*cls*, *problem_type*)

Returns whether or not an objective is defined for a problem type.

**objective_function**(*self*, *y_true*, *y_predicted*, *X=None*, *sample_weight=None*)

Objective function for precision score for multiclass classification using macro-averaging.

**positive_only**(*cls*)

If True, this objective is only valid for positive data. Defaults to False.

**score**(*self*, *y_true*, *y_predicted*, *X=None*, *sample_weight=None*)

Returns a numerical score indicating performance based on the differences between the predicted and actual values.

**Parameters**

- **y_predicted** (*pd.Series*) – Predicted values of length [n_samples]
- **y_true** (*pd.Series*) – Actual class labels of length [n_samples]
- **X** (*pd.DataFrame or np.ndarray*) – Extra data of shape [n_samples, n_features] necessary to calculate score
- **sample_weight** (*pd.DataFrame or np.ndarray*) – Sample weights used in computing objective value result

**Returns** score

**validate_inputs**(*self*, *y_true*, *y_predicted*)

Validates the input based on a few simple checks.

**Parameters**

- **y_predicted** (*pd.Series, or pd.DataFrame*) – Predicted values of length [n_samples].
- **y_true** (*pd.Series*) – Actual class labels of length [n_samples].

**Raises** **ValueError** – If the inputs are malformed.

**class** evalml.objectives.standard_metrics.**PrecisionMicro**

Precision score for multiclass classification using micro averaging.

### Example

```
>>> y_true = pd.Series([0, 1, 0, 2, 0, 1, 2, 1, 2, 0, 2])
>>> y_pred = pd.Series([0, 0, 0, 0, 1, 1, 1, 1, 2, 2, 2])
>>> np.testing.assert_almost_equal(PrecisionMicro().objective_function(y_true, y_
↪pred), 0.5454545)
```

**Attributes**

| | |
|---|---|
| **expected_range** | [0, 1] |
| **greater_is_better** | True |
| **is_bounded_like** | True |
| **name** | Precision Micro |
| **perfect_score** | 1.0 |
| **problem_types** | [ProblemTypes.MULTICLASS, ProblemTypes.TIME_SERIES_MULTICLASS] |
| **score_needs_proba** | False |

**Methods**

| | |
|---|---|
| *calculate_percent_difference* | Calculate the percent difference between scores. |
| *is_defined_for_problem_type* | Returns whether or not an objective is defined for a problem type. |
| *objective_function* | Objective function for precision score for binary classification using micro-averaging. |
| *positive_only* | If True, this objective is only valid for positive data. Defaults to False. |
| *score* | Returns a numerical score indicating performance based on the differences between the predicted and actual values. |
| *validate_inputs* | Validates the input based on a few simple checks. |

**classmethod calculate_percent_difference**(*cls*, *score*, *baseline_score*)

Calculate the percent difference between scores.

**Parameters**

- **score** (*float*) – A score. Output of the score method of this objective.
- **baseline_score** (*float*) – A score. Output of the score method of this objective. In practice, this is the score achieved on this objective with a baseline estimator.

**Returns**

**The percent difference between the scores. Note that for objectives that can be interpreted** as percentages, this will be the difference between the reference score and score. For all other objectives, the difference will be normalized by the reference score.

**Return type** float

**classmethod is_defined_for_problem_type**(*cls*, *problem_type*)

Returns whether or not an objective is defined for a problem type.

**objective_function**(*self*, *y_true*, *y_predicted*, *X=None*, *sample_weight=None*)

Objective function for precision score for binary classification using micro-averaging.

**positive_only**(*cls*)

If True, this objective is only valid for positive data. Defaults to False.

**score**(*self*, *y_true*, *y_predicted*, *X=None*, *sample_weight=None*)

Returns a numerical score indicating performance based on the differences between the predicted and actual values.

**Parameters**

- **y_predicted** (*pd.Series*) – Predicted values of length [n_samples]
- **y_true** (*pd.Series*) – Actual class labels of length [n_samples]
- **X** (*pd.DataFrame or np.ndarray*) – Extra data of shape [n_samples, n_features] necessary to calculate score
- **sample_weight** (*pd.DataFrame or np.ndarray*) – Sample weights used in computing objective value result

**Returns** score

**validate_inputs**(*self*, *y_true*, *y_predicted*)

Validates the input based on a few simple checks.

**Parameters**

- **y_predicted** (*pd.Series, or pd.DataFrame*) – Predicted values of length [n_samples].
- **y_true** (*pd.Series*) – Actual class labels of length [n_samples].

**Raises** **ValueError** – If the inputs are malformed.

**class** evalml.objectives.standard_metrics.**PrecisionWeighted**

Precision score for multiclass classification using weighted averaging.

### Example

```
>>> y_true = pd.Series([0, 1, 0, 2, 0, 1, 2, 1, 2, 0, 2])
>>> y_pred = pd.Series([0, 0, 0, 0, 1, 1, 1, 1, 2, 2, 2])
>>> np.testing.assert_almost_equal(PrecisionWeighted().objective_function(y_true, y_
↪pred), 0.5606060)
```

**Attributes**

| | |
|---|---|
| **expected_range** | [0, 1] |
| **greater_is_better** | True |
| **is_bounded_likely** | True |
| **name** | Precision Weighted |
| **perfect_score** | 1.0 |
| **problem_types** | [ProblemTypes.MULTICLASS, ProblemTypes.TIME_SERIES_MULTICLASS] |
| **score_needs_proba** | False |

**Methods**

| | |
|---|---|
| *calculate_percent_difference* | Calculate the percent difference between scores. |
| *is_defined_for_problem_type* | Returns whether or not an objective is defined for a problem type. |
| *objective_function* | Objective function for precision score for multiclass classification using weighted averaging. |
| *positive_only* | If True, this objective is only valid for positive data. Defaults to False. |
| *score* | Returns a numerical score indicating performance based on the differences between the predicted and actual values. |
| *validate_inputs* | Validates the input based on a few simple checks. |

**classmethod calculate_percent_difference**(*cls*, *score*, *baseline_score*)

Calculate the percent difference between scores.

**Parameters**

- **score** (*float*) – A score. Output of the score method of this objective.
- **baseline_score** (*float*) – A score. Output of the score method of this objective. In practice, this is the score achieved on this objective with a baseline estimator.

**Returns**

**The percent difference between the scores. Note that for objectives that can be interpreted** as percentages, this will be the difference between the reference score and score. For all other objectives, the difference will be normalized by the reference score.

**Return type** float

**classmethod is_defined_for_problem_type**(*cls*, *problem_type*)

Returns whether or not an objective is defined for a problem type.

**objective_function**(*self*, *y_true*, *y_predicted*, *X=None*, *sample_weight=None*)

Objective function for precision score for multiclass classification using weighted averaging.

**positive_only**(*cls*)

If True, this objective is only valid for positive data. Defaults to False.

**score**(*self*, *y_true*, *y_predicted*, *X=None*, *sample_weight=None*)

Returns a numerical score indicating performance based on the differences between the predicted and actual values.

**Parameters**

- **y_predicted** (*pd.Series*) – Predicted values of length [n_samples]
- **y_true** (*pd.Series*) – Actual class labels of length [n_samples]
- **X** (*pd.DataFrame or np.ndarray*) – Extra data of shape [n_samples, n_features] necessary to calculate score
- **sample_weight** (*pd.DataFrame or np.ndarray*) – Sample weights used in computing objective value result

**Returns** score

**validate_inputs**(*self*, *y_true*, *y_predicted*)

Validates the input based on a few simple checks.

**Parameters**

- **y_predicted** (*pd.Series, or pd.DataFrame*) – Predicted values of length [n_samples].
- **y_true** (*pd.Series*) – Actual class labels of length [n_samples].

**Raises** **ValueError** – If the inputs are malformed.

**class** evalml.objectives.standard_metrics.**R2**

Coefficient of determination for regression.

### Example

```
>>> y_true = pd.Series([1.5, 2, 3, 1, 0.5, 1, 2.5, 2.5, 1, 0.5, 2])
>>> y_pred = pd.Series([1.5, 2.5, 2, 1, 0.5, 1, 3, 2.25, 0.75, 0.25, 1.75])
>>> np.testing.assert_almost_equal(R2().objective_function(y_true, y_pred), 0.
↪7638036)
```

**Attributes**

| | |
|---|---|
| **expected_range** | None |
| **greater_is_better** | True |
| **is_bounded_like_percentage** | False |
| **name** | R2 |
| **perfect_score** | 1 |
| **problem_types** | [ProblemTypes.REGRESSION, ProblemTypes.TIME_SERIES_REGRESSION] |
| **score_needs_proba** | False |

**Methods**

| | |
|---|---|
| *calculate_percent_difference* | Calculate the percent difference between scores. |
| *is_defined_for_problem_type* | Returns whether or not an objective is defined for a problem type. |
| *objective_function* | Objective function for coefficient of determination for regression. |
| *positive_only* | If True, this objective is only valid for positive data. Defaults to False. |
| *score* | Returns a numerical score indicating performance based on the differences between the predicted and actual values. |
| *validate_inputs* | Validates the input based on a few simple checks. |

**classmethod calculate_percent_difference**(*cls*, *score*, *baseline_score*)

Calculate the percent difference between scores.

**Parameters**

- **score** (*float*) – A score. Output of the score method of this objective.
- **baseline_score** (*float*) – A score. Output of the score method of this objective. In practice, this is the score achieved on this objective with a baseline estimator.

**Returns**

**The percent difference between the scores. Note that for objectives that can be interpreted** as percentages, this will be the difference between the reference score and score. For all other objectives, the difference will be normalized by the reference score.

**Return type** float

**classmethod is_defined_for_problem_type**(*cls*, *problem_type*)

Returns whether or not an objective is defined for a problem type.

**objective_function**(*self*, *y_true*, *y_predicted*, *X=None*, *sample_weight=None*)

Objective function for coefficient of determination for regression.

**positive_only**(*cls*)

If True, this objective is only valid for positive data. Defaults to False.

**score**(*self*, *y_true*, *y_predicted*, *X=None*, *sample_weight=None*)

Returns a numerical score indicating performance based on the differences between the predicted and actual values.

**Parameters**

- **y_predicted** (*pd.Series*) – Predicted values of length [n_samples]
- **y_true** (*pd.Series*) – Actual class labels of length [n_samples]
- **X** (*pd.DataFrame or np.ndarray*) – Extra data of shape [n_samples, n_features] necessary to calculate score
- **sample_weight** (*pd.DataFrame or np.ndarray*) – Sample weights used in computing objective value result

**Returns** score

**validate_inputs**(*self*, *y_true*, *y_predicted*)

Validates the input based on a few simple checks.

**Parameters**

- **y_predicted** (*pd.Series, or pd.DataFrame*) – Predicted values of length [n_samples].
- **y_true** (*pd.Series*) – Actual class labels of length [n_samples].

**Raises** **ValueError** – If the inputs are malformed.

**class** evalml.objectives.standard_metrics.**Recall**

Recall score for binary classification.

### Example

```
>>> y_true = pd.Series([0, 1, 1, 0, 0, 1, 1, 1, 0, 1, 1])
>>> y_pred = pd.Series([0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1])
>>> np.testing.assert_almost_equal(Recall().objective_function(y_true, y_pred), 0.
↪1428571)
```

**Attributes**

| | |
|---|---|
| **expected_range** | [0, 1] |
| **greater_is_better** | True |
| **is_bounded_like_percentage** | True |
| **name** | Recall |
| **perfect_score** | 1.0 |
| **problem_types** | [ProblemTypes.BINARY, ProblemTypes.TIME_SERIES_BINARY] |
| **score_needs_proba** | False |

**Methods**

| | |
|---|---|
| *calculate_percent_difference* | Calculate the percent difference between scores. |
| *can_optimize_threshold* | Returns a boolean determining if we can optimize the binary classification objective threshold. |
| *decision_function* | Apply a learned threshold to predicted probabilities to get predicted classes. |
| *is_defined_for_problem_type* | Returns whether or not an objective is defined for a problem type. |
| *objective_function* | Objective function for recall score for binary classification. |
| *optimize_threshold* | Learn a binary classification threshold which optimizes the current objective. |
| *positive_only* | If True, this objective is only valid for positive data. Defaults to False. |
| *score* | Returns a numerical score indicating performance based on the differences between the predicted and actual values. |
| *validate_inputs* | Validate inputs for scoring. |

**classmethod calculate_percent_difference**(*cls*, *score*, *baseline_score*)

Calculate the percent difference between scores.

**Parameters**

- **score** (*float*) – A score. Output of the score method of this objective.
- **baseline_score** (*float*) – A score. Output of the score method of this objective. In practice, this is the score achieved on this objective with a baseline estimator.

**Returns**

**The percent difference between the scores. Note that for objectives that can be interpreted** as percentages, this will be the difference between the reference score and score. For all other objectives, the difference will be normalized by the reference score.

**Return type** float

**property can_optimize_threshold**(*cls*)

Returns a boolean determining if we can optimize the binary classification objective threshold.

This will be false for any objective that works directly with predicted probabilities, like log loss and AUC. Otherwise, it will be true.

**Returns** Whether or not an objective can be optimized.

**Return type** bool

**decision_function**(*self*, *ypred_proba*, *threshold=0.5*, *X=None*)

Apply a learned threshold to predicted probabilities to get predicted classes.

**Parameters**

- **ypred_proba** (*pd.Series, np.ndarray*) – The classifier's predicted probabilities
- **threshold** (*float, optional*) – Threshold used to make a prediction. Defaults to 0.5.
- **X** (*pd.DataFrame, optional*) – Any extra columns that are needed from training data.

**Returns** predictions

**classmethod is_defined_for_problem_type**(*cls*, *problem_type*)

Returns whether or not an objective is defined for a problem type.

**objective_function**(*self*, *y_true*, *y_predicted*, *X=None*, *sample_weight=None*)

Objective function for recall score for binary classification.

**optimize_threshold**(*self*, *ypred_proba*, *y_true*, *X=None*)

Learn a binary classification threshold which optimizes the current objective.

**Parameters**

- **ypred_proba** (*pd.Series*) – The classifier's predicted probabilities
- **y_true** (*pd.Series*) – The ground truth for the predictions.
- **X** (*pd.DataFrame, optional*) – Any extra columns that are needed from training data.

**Returns** Optimal threshold for this objective.

**Raises** **RuntimeError** – If objective cannot be optimized.

**positive_only**(*cls*)

If True, this objective is only valid for positive data. Defaults to False.

**score**(*self*, *y_true*, *y_predicted*, *X=None*, *sample_weight=None*)

Returns a numerical score indicating performance based on the differences between the predicted and actual values.

**Parameters**

- **y_predicted** (*pd.Series*) – Predicted values of length [n_samples]
- **y_true** (*pd.Series*) – Actual class labels of length [n_samples]
- **X** (*pd.DataFrame or np.ndarray*) – Extra data of shape [n_samples, n_features] necessary to calculate score
- **sample_weight** (*pd.DataFrame or np.ndarray*) – Sample weights used in computing objective value result

**Returns** score

**validate_inputs**(*self*, *y_true*, *y_predicted*)

Validate inputs for scoring.

**class** evalml.objectives.standard_metrics.**RecallMacro**

Recall score for multiclass classification using macro averaging.

### Example

```
>>> y_true = pd.Series([0, 1, 0, 2, 0, 1, 2, 1, 2, 0, 2])
>>> y_pred = pd.Series([0, 0, 0, 0, 1, 1, 1, 1, 2, 2, 2])
>>> np.testing.assert_almost_equal(RecallMacro().objective_function(y_true, y_pred),
↪ 0.5555555)
```

**Attributes**

| | |
|---|---|
| **expected_range** | [0, 1] |
| **greater_is_better** | True |
| **is_bounded_like** | True |
| **name** | Recall Macro |
| **perfect_score** | 1.0 |
| **problem_types** | [ProblemTypes.MULTICLASS, ProblemTypes.TIME_SERIES_MULTICLASS] |
| **score_needs_proba** | False |

**Methods**

| | |
|---|---|
| *calculate_percent_difference* | Calculate the percent difference between scores. |
| *is_defined_for_problem_type* | Returns whether or not an objective is defined for a problem type. |
| *objective_function* | Objective function for recall score for multiclass classification using macro-averaging. |
| *positive_only* | If True, this objective is only valid for positive data. Defaults to False. |
| *score* | Returns a numerical score indicating performance based on the differences between the predicted and actual values. |
| *validate_inputs* | Validates the input based on a few simple checks. |

**classmethod calculate_percent_difference**(*cls*, *score*, *baseline_score*)

Calculate the percent difference between scores.

**Parameters**

- **score** (*float*) – A score. Output of the score method of this objective.
- **baseline_score** (*float*) – A score. Output of the score method of this objective. In practice, this is the score achieved on this objective with a baseline estimator.

**Returns**

**The percent difference between the scores. Note that for objectives that can be interpreted** as percentages, this will be the difference between the reference score and score. For all other objectives, the difference will be normalized by the reference score.

**Return type** float

**classmethod is_defined_for_problem_type**(*cls*, *problem_type*)

Returns whether or not an objective is defined for a problem type.

**objective_function**(*self*, *y_true*, *y_predicted*, *X=None*, *sample_weight=None*)

Objective function for recall score for multiclass classification using macro-averaging.

**positive_only**(*cls*)

If True, this objective is only valid for positive data. Defaults to False.

**score**(*self*, *y_true*, *y_predicted*, *X=None*, *sample_weight=None*)

Returns a numerical score indicating performance based on the differences between the predicted and actual values.

**Parameters**

- **y_predicted** (*pd.Series*) – Predicted values of length [n_samples]
- **y_true** (*pd.Series*) – Actual class labels of length [n_samples]
- **X** (*pd.DataFrame or np.ndarray*) – Extra data of shape [n_samples, n_features] necessary to calculate score
- **sample_weight** (*pd.DataFrame or np.ndarray*) – Sample weights used in computing objective value result

**Returns** score

**validate_inputs**(*self*, *y_true*, *y_predicted*)

Validates the input based on a few simple checks.

**Parameters**

- **y_predicted** (*pd.Series, or pd.DataFrame*) – Predicted values of length [n_samples].
- **y_true** (*pd.Series*) – Actual class labels of length [n_samples].

**Raises** **ValueError** – If the inputs are malformed.

**class** evalml.objectives.standard_metrics.**RecallMicro**

Recall score for multiclass classification using micro averaging.

### Example

```
>>> y_true = pd.Series([0, 1, 0, 2, 0, 1, 2, 1, 2, 0, 2])
>>> y_pred = pd.Series([0, 0, 0, 0, 1, 1, 1, 1, 2, 2, 2])
>>> np.testing.assert_almost_equal(RecallMicro().objective_function(y_true, y_pred),
↪ 0.5454545)
```

**Attributes**

| | |
|---|---|
| **expected_range** | [0, 1] |
| **greater_is_better** | True |
| **is_bounded_like** | True |
| **name** | Recall Micro |
| **perfect_score** | 1.0 |
| **problem_types** | [ProblemTypes.MULTICLASS, ProblemTypes.TIME_SERIES_MULTICLASS] |
| **score_needs_proba** | False |

**Methods**

| | |
|---|---|
| *calculate_percent_difference* | Calculate the percent difference between scores. |
| *is_defined_for_problem_type* | Returns whether or not an objective is defined for a problem type. |
| *objective_function* | Objective function for recall score for multiclass classification using micro-averaging. |
| *positive_only* | If True, this objective is only valid for positive data. Defaults to False. |
| *score* | Returns a numerical score indicating performance based on the differences between the predicted and actual values. |
| *validate_inputs* | Validates the input based on a few simple checks. |

**classmethod calculate_percent_difference**(*cls*, *score*, *baseline_score*)

Calculate the percent difference between scores.

**Parameters**

- **score** (*float*) – A score. Output of the score method of this objective.
- **baseline_score** (*float*) – A score. Output of the score method of this objective. In practice, this is the score achieved on this objective with a baseline estimator.

**Returns**

**The percent difference between the scores. Note that for objectives that can be interpreted** as percentages, this will be the difference between the reference score and score. For all other objectives, the difference will be normalized by the reference score.

**Return type** float

**classmethod is_defined_for_problem_type**(*cls*, *problem_type*)

Returns whether or not an objective is defined for a problem type.

**objective_function**(*self*, *y_true*, *y_predicted*, *X=None*, *sample_weight=None*)

Objective function for recall score for multiclass classification using micro-averaging.

**positive_only**(*cls*)

If True, this objective is only valid for positive data. Defaults to False.

**score**(*self*, *y_true*, *y_predicted*, *X=None*, *sample_weight=None*)

Returns a numerical score indicating performance based on the differences between the predicted and actual values.

**Parameters**

- **y_predicted** (*pd.Series*) – Predicted values of length [n_samples]
- **y_true** (*pd.Series*) – Actual class labels of length [n_samples]
- **X** (*pd.DataFrame or np.ndarray*) – Extra data of shape [n_samples, n_features] necessary to calculate score
- **sample_weight** (*pd.DataFrame or np.ndarray*) – Sample weights used in computing objective value result

**Returns** score

**validate_inputs**(*self*, *y_true*, *y_predicted*)

Validates the input based on a few simple checks.

**Parameters**

- **y_predicted** (*pd.Series, or pd.DataFrame*) – Predicted values of length [n_samples].
- **y_true** (*pd.Series*) – Actual class labels of length [n_samples].

**Raises** **ValueError** – If the inputs are malformed.

**class** evalml.objectives.standard_metrics.**RecallWeighted**

Recall score for multiclass classification using weighted averaging.

### Example

```
>>> y_true = pd.Series([0, 1, 0, 2, 0, 1, 2, 1, 2, 0, 2])
>>> y_pred = pd.Series([0, 0, 0, 0, 1, 1, 1, 1, 2, 2, 2])
>>> np.testing.assert_almost_equal(RecallWeighted().objective_function(y_true, y_
↪pred), 0.5454545)
```

**Attributes**

| | |
|---|---|
| **expected_range** | [0, 1] |
| **greater_is_better** | True |
| **is_bounded_like_percentage** | True |
| **name** | Recall Weighted |
| **perfect_score** | 1.0 |
| **problem_types** | [ProblemTypes.MULTICLASS, ProblemTypes.TIME_SERIES_MULTICLASS] |
| **score_needs_proba** | False |

**Methods**

| | |
|---|---|
| *calculate_percent_difference* | Calculate the percent difference between scores. |
| *is_defined_for_problem_type* | Returns whether or not an objective is defined for a problem type. |
| *objective_function* | Objective function for recall score for multiclass classification using weighted averaging. |
| *positive_only* | If True, this objective is only valid for positive data. Defaults to False. |
| *score* | Returns a numerical score indicating performance based on the differences between the predicted and actual values. |
| *validate_inputs* | Validates the input based on a few simple checks. |

**classmethod calculate_percent_difference**(*cls*, *score*, *baseline_score*)

Calculate the percent difference between scores.

**Parameters**

- **score** (*float*) – A score. Output of the score method of this objective.
- **baseline_score** (*float*) – A score. Output of the score method of this objective. In practice, this is the score achieved on this objective with a baseline estimator.

**Returns**

**The percent difference between the scores. Note that for objectives that can be interpreted** as percentages, this will be the difference between the reference score and score. For all other objectives, the difference will be normalized by the reference score.

**Return type** float

**classmethod is_defined_for_problem_type**(*cls*, *problem_type*)

Returns whether or not an objective is defined for a problem type.

**objective_function**(*self*, *y_true*, *y_predicted*, *X=None*, *sample_weight=None*)

Objective function for recall score for multiclass classification using weighted averaging.

**positive_only**(*cls*)

If True, this objective is only valid for positive data. Defaults to False.

**score**(*self*, *y_true*, *y_predicted*, *X=None*, *sample_weight=None*)

Returns a numerical score indicating performance based on the differences between the predicted and actual values.

**Parameters**

- **y_predicted** (*pd.Series*) – Predicted values of length [n_samples]
- **y_true** (*pd.Series*) – Actual class labels of length [n_samples]
- **X** (*pd.DataFrame or np.ndarray*) – Extra data of shape [n_samples, n_features] necessary to calculate score
- **sample_weight** (*pd.DataFrame or np.ndarray*) – Sample weights used in computing objective value result

**Returns** score

**validate_inputs**(*self*, *y_true*, *y_predicted*)

Validates the input based on a few simple checks.

**Parameters**

- **y_predicted** (*pd.Series, or pd.DataFrame*) – Predicted values of length [n_samples].
- **y_true** (*pd.Series*) – Actual class labels of length [n_samples].

**Raises** **ValueError** – If the inputs are malformed.

**class** evalml.objectives.standard_metrics.**RootMeanSquaredError**

Root mean squared error for regression.

### Example

```
>>> y_true = pd.Series([1.5, 2, 3, 1, 0.5, 1, 2.5, 2.5, 1, 0.5, 2])
>>> y_pred = pd.Series([1.5, 2.5, 2, 1, 0.5, 1, 3, 2.25, 0.75, 0.25, 1.75])
>>> np.testing.assert_almost_equal(RootMeanSquaredError().objective_function(y_true,
↪ y_pred), 0.3988620)
```

**Attributes**

| | |
|---|---|
| **expected_range** | None |
| **greater_is_better** | False |
| **is_bounded_like_a_percentage** | False |
| **name** | Root Mean Squared Error |
| **perfect_score** | 0.0 |
| **problem_types** | [ProblemTypes.REGRESSION, ProblemTypes.TIME_SERIES_REGRESSION] |
| **score_needs_proba** | False |

**Methods**

| | |
|---|---|
| *calculate_percent_difference* | Calculate the percent difference between scores. |
| *is_defined_for_problem_type* | Returns whether or not an objective is defined for a problem type. |
| *objective_function* | Objective function for root mean squared error for regression. |
| *positive_only* | If True, this objective is only valid for positive data. Defaults to False. |
| *score* | Returns a numerical score indicating performance based on the differences between the predicted and actual values. |
| *validate_inputs* | Validates the input based on a few simple checks. |

**classmethod calculate_percent_difference**(*cls*, *score*, *baseline_score*)

Calculate the percent difference between scores.

**Parameters**

- **score** (*float*) – A score. Output of the score method of this objective.
- **baseline_score** (*float*) – A score. Output of the score method of this objective. In practice, this is the score achieved on this objective with a baseline estimator.

**Returns**

**The percent difference between the scores. Note that for objectives that can be interpreted** as percentages, this will be the difference between the reference score and score. For all other objectives, the difference will be normalized by the reference score.

**Return type** float

**classmethod is_defined_for_problem_type**(*cls*, *problem_type*)

Returns whether or not an objective is defined for a problem type.

**objective_function**(*self*, *y_true*, *y_predicted*, *X=None*, *sample_weight=None*)

Objective function for root mean squared error for regression.

**positive_only**(*cls*)

If True, this objective is only valid for positive data. Defaults to False.

**score**(*self*, *y_true*, *y_predicted*, *X=None*, *sample_weight=None*)

Returns a numerical score indicating performance based on the differences between the predicted and actual values.

**Parameters**

- **y_predicted** (*pd.Series*) – Predicted values of length [n_samples]
- **y_true** (*pd.Series*) – Actual class labels of length [n_samples]
- **X** (*pd.DataFrame or np.ndarray*) – Extra data of shape [n_samples, n_features] necessary to calculate score
- **sample_weight** (*pd.DataFrame or np.ndarray*) – Sample weights used in computing objective value result

**Returns** score

**validate_inputs**(*self*, *y_true*, *y_predicted*)

Validates the input based on a few simple checks.

**Parameters**

- **y_predicted** (*pd.Series, or pd.DataFrame*) – Predicted values of length [n_samples].
- **y_true** (*pd.Series*) – Actual class labels of length [n_samples].

**Raises** **ValueError** – If the inputs are malformed.

**class** evalml.objectives.standard_metrics.**RootMeanSquaredLogError**

Root mean squared log error for regression.

Only valid for nonnegative inputs. Otherwise, will throw a ValueError.

### Example

```
>>> y_true = pd.Series([1.5, 2, 3, 1, 0.5, 1, 2.5, 2.5, 1, 0.5, 2])
>>> y_pred = pd.Series([1.5, 2.5, 2, 1, 0.5, 1, 3, 2.25, 0.75, 0.25, 1.75])
>>> np.testing.assert_almost_equal(RootMeanSquaredLogError().objective_function(y_
↪true, y_pred), 0.13090204)
```

**Attributes**

| | |
|---|---|
| **expected_range** | None |
| **greater_is_better** | False |
| **is_bounded_like_a_percentage** | False |
| **name** | Root Mean Squared Log Error |
| **perfect_score** | 0.0 |
| **problem_types** | [ProblemTypes.REGRESSION, ProblemTypes.TIME_SERIES_REGRESSION] |
| **score_needs_proba** | False |

**Methods**

| | |
|---|---|
| *calculate_percent_difference* | Calculate the percent difference between scores. |
| *is_defined_for_problem_type* | Returns whether or not an objective is defined for a problem type. |
| *objective_function* | Objective function for root mean squared log error for regression. |
| *positive_only* | If True, this objective is only valid for positive data. |
| *score* | Returns a numerical score indicating performance based on the differences between the predicted and actual values. |
| *validate_inputs* | Validates the input based on a few simple checks. |

**classmethod calculate_percent_difference**(*cls*, *score*, *baseline_score*)

Calculate the percent difference between scores.

**Parameters**

- **score** (*float*) – A score. Output of the score method of this objective.

- **baseline_score** (*float*) – A score. Output of the score method of this objective. In practice, this is the score achieved on this objective with a baseline estimator.

**Returns**

**The percent difference between the scores. Note that for objectives that can be interpreted** as percentages, this will be the difference between the reference score and score. For all other objectives, the difference will be normalized by the reference score.

**Return type** float

**classmethod is_defined_for_problem_type**(*cls*, *problem_type*)

Returns whether or not an objective is defined for a problem type.

**objective_function**(*self*, *y_true*, *y_predicted*, *X=None*, *sample_weight=None*)

Objective function for root mean squared log error for regression.

**positive_only**(*self*)

If True, this objective is only valid for positive data.

**score**(*self*, *y_true*, *y_predicted*, *X=None*, *sample_weight=None*)

Returns a numerical score indicating performance based on the differences between the predicted and actual values.

**Parameters**

- **y_predicted** (*pd.Series*) – Predicted values of length [n_samples]
- **y_true** (*pd.Series*) – Actual class labels of length [n_samples]
- **X** (*pd.DataFrame or np.ndarray*) – Extra data of shape [n_samples, n_features] necessary to calculate score
- **sample_weight** (*pd.DataFrame or np.ndarray*) – Sample weights used in computing objective value result

**Returns** score

**validate_inputs**(*self*, *y_true*, *y_predicted*)

Validates the input based on a few simple checks.

**Parameters**

- **y_predicted** (*pd.Series, or pd.DataFrame*) – Predicted values of length [n_samples].
- **y_true** (*pd.Series*) – Actual class labels of length [n_samples].

**Raises** **ValueError** – If the inputs are malformed.

### time_series_regression_objective

Base class for all time series regression objectives.

## Module Contents

## Classes Summary

| | |
|---|---|
| *TimeSeriesRegressionObjective* | Base class for all time series regression objectives. |

## Contents

**class** evalml.objectives.time_series_regression_objective.**TimeSeriesRegressionObjective**

Base class for all time series regression objectives.

**Attributes**

| | |
|---|---|
| **problem_types** | [ProblemTypes.TIME_SERIES_REGRESSION] |

**Methods**

| | |
|---|---|
| *calculate_percent_difference* | Calculate the percent difference between scores. |
| *expected_range* | Returns the expected range of the objective, which is not necessarily the possible ranges. |
| *greater_is_better* | Returns a boolean determining if a greater score indicates better model performance. |
| *is_bounded_like_percentage* | Returns whether this objective is bounded between 0 and 1, inclusive. |
| *is_defined_for_problem_type* | Returns whether or not an objective is defined for a problem type. |
| *name* | Returns a name describing the objective. |
| *objective_function* | Computes the relative value of the provided predictions compared to the actual labels, according a specified metric. |
| *perfect_score* | Returns the score obtained by evaluating this objective on a perfect model. |
| *positive_only* | If True, this objective is only valid for positive data. Defaults to False. |
| *score* | Returns a numerical score indicating performance based on the differences between the predicted and actual values. |
| *score_needs_proba* | Returns a boolean determining if the score() method needs probability estimates. |
| *validate_inputs* | Validates the input based on a few simple checks. |

**classmethod calculate_percent_difference**(*cls*, *score*, *baseline_score*)

Calculate the percent difference between scores.

**Parameters**

- **score** (*float*) – A score. Output of the score method of this objective.
- **baseline_score** (*float*) – A score. Output of the score method of this objective. In practice, this is the score achieved on this objective with a baseline estimator.

**Returns**

**The percent difference between the scores. Note that for objectives that can be interpreted** as percentages, this will be the difference between the reference score and score. For all other objectives, the difference will be normalized by the reference score.

**Return type** float

**property expected_range**(*cls*)

Returns the expected range of the objective, which is not necessarily the possible ranges.

For example, our expected R2 range is from [-1, 1], although the actual range is (-inf, 1].

**property greater_is_better**(*cls*)

Returns a boolean determining if a greater score indicates better model performance.

**property is_bounded_like_percentage**(*cls*)

Returns whether this objective is bounded between 0 and 1, inclusive.

**classmethod is_defined_for_problem_type**(*cls*, *problem_type*)

Returns whether or not an objective is defined for a problem type.

**property name**(*cls*)

Returns a name describing the objective.

**abstract classmethod objective_function**(*cls*, *y_true*, *y_predicted*, *X=None*, *sample_weight=None*)

Computes the relative value of the provided predictions compared to the actual labels, according a specified metric.

**Parameters**

- **y_predicted** (*pd.Series*) – Predicted values of length [n_samples]
- **y_true** (*pd.Series*) – Actual class labels of length [n_samples]
- **X** (*pd.DataFrame or np.ndarray*) – Extra data of shape [n_samples, n_features] necessary to calculate score
- **sample_weight** (*pd.DataFrame or np.ndarray*) – Sample weights used in computing objective value result

**Returns** Numerical value used to calculate score

**property perfect_score**(*cls*)

Returns the score obtained by evaluating this objective on a perfect model.

**positive_only**(*cls*)

If True, this objective is only valid for positive data. Defaults to False.

**score**(*self*, *y_true*, *y_predicted*, *X=None*, *sample_weight=None*)

Returns a numerical score indicating performance based on the differences between the predicted and actual values.

**Parameters**

- **y_predicted** (*pd.Series*) – Predicted values of length [n_samples]
- **y_true** (*pd.Series*) – Actual class labels of length [n_samples]
- **X** (*pd.DataFrame or np.ndarray*) – Extra data of shape [n_samples, n_features] necessary to calculate score

- **sample_weight** (*pd.DataFrame or np.ndarray*) – Sample weights used in computing objective value result

**Returns** score

**property score_needs_proba**(*cls*)

Returns a boolean determining if the score() method needs probability estimates.

This should be true for objectives which work with predicted probabilities, like log loss or AUC, and false for objectives which compare predicted class labels to the actual labels, like F1 or correlation.

**validate_inputs**(*self*, *y_true*, *y_predicted*)

Validates the input based on a few simple checks.

**Parameters**

- **y_predicted** (*pd.Series, or pd.DataFrame*) – Predicted values of length [n_samples].
- **y_true** (*pd.Series*) – Actual class labels of length [n_samples].

**Raises** **ValueError** – If the inputs are malformed.

### utils

Utility methods for EvalML objectives.

### Module Contents

### Functions

| | |
|---|---|
| *get_all_objective_names* | Get a list of the names of all objectives. |
| *get_core_objective_names* | Get a list of all valid core objectives. |
| *get_core_objectives* | Returns all core objective instances associated with the given problem type. |
| *get_non_core_objectives* | Get non-core objective classes. |
| *get_objective* | Returns the Objective class corresponding to a given objective name. |
| *get_optimization_objectives* | Get objectives for optimization. |
| *get_ranking_objectives* | Get objectives for pipeline rankings. |
| *ranking_only_objectives* | Get ranking-only objective classes. |

### Contents

evalml.objectives.utils.**get_all_objective_names**()

Get a list of the names of all objectives.

**Returns** Objective names

**Return type** list (str)

`evalml.objectives.utils.get_core_objective_names()`

Get a list of all valid core objectives.

**Returns** Objective names.

**Return type** list[str]

`evalml.objectives.utils.get_core_objectives(problem_type)`

Returns all core objective instances associated with the given problem type.

Core objectives are designed to work out-of-the-box for any dataset.

**Parameters** **problem_type** (*str/ProblemTypes*) – Type of problem

**Returns** List of ObjectiveBase instances

### Examples

```
>>> for objective in get_core_objectives("regression"):
...     print(objective.name)
ExpVariance
MaxError
MedianAE
MSE
MAE
R2
Root Mean Squared Error
>>> for objective in get_core_objectives("binary"):
...     print(objective.name)
MCC Binary
Log Loss Binary
Gini
AUC
Precision
F1
Balanced Accuracy Binary
Accuracy Binary
```

`evalml.objectives.utils.get_non_core_objectives()`

Get non-core objective classes.

Non-core objectives are objectives that are domain-specific. Users typically need to configure these objectives before using them in AutoMLSearch.

**Returns** List of ObjectiveBase classes

`evalml.objectives.utils.get_objective(objective, return_instance=False, **kwargs)`

Returns the Objective class corresponding to a given objective name.

**Parameters**

- **objective** (*str or ObjectiveBase*) – Name or instance of the objective class.
- **return_instance** (*bool*) – Whether to return an instance of the objective. This only applies if objective is of type str. Note that the instance will be initialized with default arguments.
- **kwargs** (*Any*) – Any keyword arguments to pass into the objective. Only used when return_instance=True.

**Returns** ObjectiveBase if the parameter objective is of type ObjectiveBase. If objective is instead a valid objective name, function will return the class corresponding to that name. If return_instance is True, an instance of that objective will be returned.

**Raises**

- **TypeError** – If objective is None.
- **TypeError** – If objective is not a string and not an instance of ObjectiveBase.
- **ObjectiveNotFoundError** – If input objective is not a valid objective.
- **ObjectiveCreationError** – If objective cannot be created properly.

`evalml.objectives.utils.get_optimization_objectives(problem_type)`

Get objectives for optimization.

**Parameters** **problem_type** (*str/ProblemTypes*) – Type of problem

**Returns** List of ObjectiveBase instances

`evalml.objectives.utils.get_ranking_objectives(problem_type)`

Get objectives for pipeline rankings.

**Parameters** **problem_type** (*str/ProblemTypes*) – Type of problem

**Returns** List of ObjectiveBase instances

`evalml.objectives.utils.ranking_only_objectives()`

Get ranking-only objective classes.

Ranking-only objectives are objectives that are useful for evaluating the performance of a model, but should not be used as an optimization objective during AutoMLSearch for various reasons.

**Returns** List of ObjectiveBase classes

## Package Contents

### Classes Summary

| | |
|---|---|
| *AccuracyBinary* | Accuracy score for binary classification. |
| *AccuracyMulticlass* | Accuracy score for multiclass classification. |
| *AUC* | AUC score for binary classification. |
| *AUCMacro* | AUC score for multiclass classification using macro averaging. |
| *AUCMicro* | AUC score for multiclass classification using micro averaging. |
| *AUCWeighted* | AUC Score for multiclass classification using weighted averaging. |
| *BalancedAccuracyBinary* | Balanced accuracy score for binary classification. |
| *BalancedAccuracyMulticlass* | Balanced accuracy score for multiclass classification. |
| *BinaryClassificationObjective* | Base class for all binary classification objectives. |

continues on next page

Table 4 – continued from previous page

| | |
|---|---|
| *CostBenefitMatrix* | Score using a cost-benefit matrix. Scores quantify the benefits of a given value, so greater numeric scores represents a better score. Costs and scores can be negative, indicating that a value is not beneficial. For example, in the case of monetary profit, a negative cost and/or score represents loss of cash flow. |
| *ExpVariance* | Explained variance score for regression. |
| *F1* | F1 score for binary classification. |
| *F1Macro* | F1 score for multiclass classification using macro averaging. |
| *F1Micro* | F1 score for multiclass classification using micro averaging. |
| *F1Weighted* | F1 score for multiclass classification using weighted averaging. |
| *FraudCost* | Score the percentage of money lost of the total transaction amount process due to fraud. |
| *Gini* | Gini coefficient for binary classification. |
| *LeadScoring* | Lead scoring. |
| *LogLossBinary* | Log Loss for binary classification. |
| *LogLossMulticlass* | Log Loss for multiclass classification. |
| *MAE* | Mean absolute error for regression. |
| *MAPE* | Mean absolute percentage error for time series regression. Scaled by 100 to return a percentage. |
| *MaxError* | Maximum residual error for regression. |
| *MCCBinary* | Matthews correlation coefficient for binary classification. |
| *MCCMulticlass* | Matthews correlation coefficient for multiclass classification. |
| *MeanSquaredLogError* | Mean squared log error for regression. |
| *MedianAE* | Median absolute error for regression. |
| *MSE* | Mean squared error for regression. |
| *MulticlassClassificationObjective* | Base class for all multiclass classification objectives. |
| *ObjectiveBase* | Base class for all objectives. |
| *Precision* | Precision score for binary classification. |
| *PrecisionMacro* | Precision score for multiclass classification using macro-averaging. |
| *PrecisionMicro* | Precision score for multiclass classification using micro averaging. |
| *PrecisionWeighted* | Precision score for multiclass classification using weighted averaging. |
| *R2* | Coefficient of determination for regression. |
| *Recall* | Recall score for binary classification. |
| *RecallMacro* | Recall score for multiclass classification using macro averaging. |
| *RecallMicro* | Recall score for multiclass classification using micro averaging. |
| *RecallWeighted* | Recall score for multiclass classification using weighted averaging. |
| *RegressionObjective* | Base class for all regression objectives. |
| *RootMeanSquaredError* | Root mean squared error for regression. |
| *RootMeanSquaredLogError* | Root mean squared log error for regression. |

continues on next page

Table 4 – continued from previous page

| | |
|---|---|
| *SensitivityLowAlert* | Create instance of SensitivityLowAlert. |

### Functions

| | |
|---|---|
| *get_all_objective_names* | Get a list of the names of all objectives. |
| *get_core_objective_names* | Get a list of all valid core objectives. |
| *get_core_objectives* | Returns all core objective instances associated with the given problem type. |
| *get_non_core_objectives* | Get non-core objective classes. |
| *get_objective* | Returns the Objective class corresponding to a given objective name. |
| *get_optimization_objectives* | Get objectives for optimization. |
| *get_ranking_objectives* | Get objectives for pipeline rankings. |
| *ranking_only_objectives* | Get ranking-only objective classes. |

### Contents

**class** evalml.objectives.**AccuracyBinary**

Accuracy score for binary classification.

#### Example

```
>>> y_true = pd.Series([0, 0, 0, 0, 0, 1, 1, 1, 0, 1, 1])
>>> y_pred = pd.Series([0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1])
>>> np.testing.assert_almost_equal(AccuracyBinary().objective_function(y_true, y_
↪pred), 0.6363636)
```

**Attributes**

| | |
|---|---|
| **expected_range** | [0, 1] |
| **greater_is_better** | True |
| **is_bounded_likely** | True |
| **name** | Accuracy Binary |
| **perfect_score** | 1.0 |
| **problem_types** | [ProblemTypes.BINARY, ProblemTypes.TIME_SERIES_BINARY] |
| **score_needs_proba** | False |

**Methods**

| | |
|---|---|
| *calculate_percent_difference* | Calculate the percent difference between scores. |
| *can_optimize_threshold* | Returns a boolean determining if we can optimize the binary classification objective threshold. |
| *decision_function* | Apply a learned threshold to predicted probabilities to get predicted classes. |
| *is_defined_for_problem_type* | Returns whether or not an objective is defined for a problem type. |
| *objective_function* | Objective function for accuracy score for binary classification. |
| *optimize_threshold* | Learn a binary classification threshold which optimizes the current objective. |
| *positive_only* | If True, this objective is only valid for positive data. Defaults to False. |
| *score* | Returns a numerical score indicating performance based on the differences between the predicted and actual values. |
| *validate_inputs* | Validate inputs for scoring. |

**classmethod calculate_percent_difference**(*cls*, *score*, *baseline_score*)

Calculate the percent difference between scores.

**Parameters**

- **score** (*float*) – A score. Output of the score method of this objective.
- **baseline_score** (*float*) – A score. Output of the score method of this objective. In practice, this is the score achieved on this objective with a baseline estimator.

**Returns**

**The percent difference between the scores. Note that for objectives that can be interpreted** as percentages, this will be the difference between the reference score and score. For all other objectives, the difference will be normalized by the reference score.

**Return type** float

**property can_optimize_threshold**(*cls*)

Returns a boolean determining if we can optimize the binary classification objective threshold.

This will be false for any objective that works directly with predicted probabilities, like log loss and AUC. Otherwise, it will be true.

**Returns** Whether or not an objective can be optimized.

**Return type** bool

**decision_function**(*self*, *ypred_proba*, *threshold=0.5*, *X=None*)

Apply a learned threshold to predicted probabilities to get predicted classes.

**Parameters**

- **ypred_proba** (*pd.Series, np.ndarray*) – The classifier's predicted probabilities
- **threshold** (*float, optional*) – Threshold used to make a prediction. Defaults to 0.5.
- **X** (*pd.DataFrame, optional*) – Any extra columns that are needed from training data.

**Returns** predictions

**classmethod is_defined_for_problem_type**(*cls*, *problem_type*)

Returns whether or not an objective is defined for a problem type.

**objective_function**(*self*, *y_true*, *y_predicted*, *X=None*, *sample_weight=None*)

Objective function for accuracy score for binary classification.

**optimize_threshold**(*self*, *ypred_proba*, *y_true*, *X=None*)

Learn a binary classification threshold which optimizes the current objective.

**Parameters**

- **ypred_proba** (*pd.Series*) – The classifier's predicted probabilities
- **y_true** (*pd.Series*) – The ground truth for the predictions.
- **X** (*pd.DataFrame, optional*) – Any extra columns that are needed from training data.

**Returns** Optimal threshold for this objective.

**Raises** **RuntimeError** – If objective cannot be optimized.

**positive_only**(*cls*)

If True, this objective is only valid for positive data. Defaults to False.

**score**(*self*, *y_true*, *y_predicted*, *X=None*, *sample_weight=None*)

Returns a numerical score indicating performance based on the differences between the predicted and actual values.

**Parameters**

- **y_predicted** (*pd.Series*) – Predicted values of length [n_samples]
- **y_true** (*pd.Series*) – Actual class labels of length [n_samples]
- **X** (*pd.DataFrame or np.ndarray*) – Extra data of shape [n_samples, n_features] necessary to calculate score
- **sample_weight** (*pd.DataFrame or np.ndarray*) – Sample weights used in computing objective value result

**Returns** score

**validate_inputs**(*self*, *y_true*, *y_predicted*)

Validate inputs for scoring.

**class** evalml.objectives.**AccuracyMulticlass**

Accuracy score for multiclass classification.

### Example

```
>>> y_true = pd.Series([0, 1, 0, 2, 0, 1, 2, 1, 2, 0, 2])
>>> y_pred = pd.Series([0, 0, 0, 0, 1, 1, 1, 1, 2, 2, 2])
>>> np.testing.assert_almost_equal(AccuracyMulticlass().objective_function(y_true,␣
↪y_pred), 0.5454545)
```

**Attributes**

| | |
|---|---|
| **expected_range** | [0, 1] |
| **greater_is_better** | True |
| **is_bounded_like_percentage** | True |
| **name** | Accuracy Multiclass |
| **perfect_score** | 1.0 |
| **problem_types** | [ProblemTypes.MULTICLASS, ProblemTypes.TIME_SERIES_MULTICLASS] |
| **score_needs_proba** | False |

**Methods**

| | |
|---|---|
| *calculate_percent_difference* | Calculate the percent difference between scores. |
| *is_defined_for_problem_type* | Returns whether or not an objective is defined for a problem type. |
| *objective_function* | Objective function for accuracy score for multiclass classification. |
| *positive_only* | If True, this objective is only valid for positive data. Defaults to False. |
| *score* | Returns a numerical score indicating performance based on the differences between the predicted and actual values. |
| *validate_inputs* | Validates the input based on a few simple checks. |

**classmethod calculate_percent_difference**(*cls*, *score*, *baseline_score*)

Calculate the percent difference between scores.

**Parameters**

- **score** (*float*) – A score. Output of the score method of this objective.
- **baseline_score** (*float*) – A score. Output of the score method of this objective. In practice, this is the score achieved on this objective with a baseline estimator.

**Returns**

**The percent difference between the scores. Note that for objectives that can be interpreted** as percentages, this will be the difference between the reference score and score. For all other objectives, the difference will be normalized by the reference score.

**Return type** float

**classmethod is_defined_for_problem_type**(*cls*, *problem_type*)

Returns whether or not an objective is defined for a problem type.

**objective_function**(*self*, *y_true*, *y_predicted*, *X=None*, *sample_weight=None*)

Objective function for accuracy score for multiclass classification.

**positive_only**(*cls*)

If True, this objective is only valid for positive data. Defaults to False.

**score**(*self*, *y_true*, *y_predicted*, *X=None*, *sample_weight=None*)

Returns a numerical score indicating performance based on the differences between the predicted and actual values.

**Parameters**

- **y_predicted** (*pd.Series*) – Predicted values of length [n_samples]
- **y_true** (*pd.Series*) – Actual class labels of length [n_samples]
- **X** (*pd.DataFrame or np.ndarray*) – Extra data of shape [n_samples, n_features] necessary to calculate score
- **sample_weight** (*pd.DataFrame or np.ndarray*) – Sample weights used in computing objective value result

**Returns** score

**validate_inputs**(*self*, *y_true*, *y_predicted*)

Validates the input based on a few simple checks.

**Parameters**

- **y_predicted** (*pd.Series, or pd.DataFrame*) – Predicted values of length [n_samples].
- **y_true** (*pd.Series*) – Actual class labels of length [n_samples].

**Raises** **ValueError** – If the inputs are malformed.

**class** evalml.objectives.**AUC**

AUC score for binary classification.

### Example

```
>>> y_true = pd.Series([0, 1, 1, 0, 0, 1, 1, 1, 0, 1, 1])
>>> y_pred = pd.Series([0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1])
>>> np.testing.assert_almost_equal(AUC().objective_function(y_true, y_pred), 0.
↪5714285)
```

**Attributes**

| | |
|---|---|
| **expected_range** | [0, 1] |
| **greater_is_better** | True |
| **is_bounded_likely** | True |
| **name** | AUC |
| **perfect_score** | 1.0 |
| **problem_types** | [ProblemTypes.BINARY, ProblemTypes.TIME_SERIES_BINARY] |
| **score_needs_proba** | True |

**Methods**

| | |
|---|---|
| *calculate_percent_difference* | Calculate the percent difference between scores. |
| *can_optimize_threshold* | Returns a boolean determining if we can optimize the binary classification objective threshold. |
| *decision_function* | Apply a learned threshold to predicted probabilities to get predicted classes. |
| *is_defined_for_problem_type* | Returns whether or not an objective is defined for a problem type. |
| *objective_function* | Objective function for AUC score for binary classification. |
| *optimize_threshold* | Learn a binary classification threshold which optimizes the current objective. |
| *positive_only* | If True, this objective is only valid for positive data. Defaults to False. |
| *score* | Returns a numerical score indicating performance based on the differences between the predicted and actual values. |
| *validate_inputs* | Validate inputs for scoring. |

**classmethod calculate_percent_difference**(*cls*, *score*, *baseline_score*)

Calculate the percent difference between scores.

**Parameters**

- **score** (*float*) – A score. Output of the score method of this objective.
- **baseline_score** (*float*) – A score. Output of the score method of this objective. In practice, this is the score achieved on this objective with a baseline estimator.

**Returns**

**The percent difference between the scores. Note that for objectives that can be interpreted** as percentages, this will be the difference between the reference score and score. For all other objectives, the difference will be normalized by the reference score.

**Return type** float

**property can_optimize_threshold**(*cls*)

Returns a boolean determining if we can optimize the binary classification objective threshold.

This will be false for any objective that works directly with predicted probabilities, like log loss and AUC. Otherwise, it will be true.

**Returns** Whether or not an objective can be optimized.

**Return type** bool

**decision_function**(*self*, *ypred_proba*, *threshold=0.5*, *X=None*)

Apply a learned threshold to predicted probabilities to get predicted classes.

**Parameters**

- **ypred_proba** (*pd.Series, np.ndarray*) – The classifier's predicted probabilities
- **threshold** (*float, optional*) – Threshold used to make a prediction. Defaults to 0.5.
- **X** (*pd.DataFrame, optional*) – Any extra columns that are needed from training data.

**Returns** predictions

**classmethod is_defined_for_problem_type**(*cls*, *problem_type*)

Returns whether or not an objective is defined for a problem type.

**objective_function**(*self*, *y_true*, *y_predicted*, *X=None*, *sample_weight=None*)

Objective function for AUC score for binary classification.

**optimize_threshold**(*self*, *ypred_proba*, *y_true*, *X=None*)

Learn a binary classification threshold which optimizes the current objective.

**Parameters**

- **ypred_proba** (*pd.Series*) – The classifier's predicted probabilities
- **y_true** (*pd.Series*) – The ground truth for the predictions.
- **X** (*pd.DataFrame, optional*) – Any extra columns that are needed from training data.

**Returns** Optimal threshold for this objective.

**Raises** **RuntimeError** – If objective cannot be optimized.

**positive_only**(*cls*)

If True, this objective is only valid for positive data. Defaults to False.

**score**(*self*, *y_true*, *y_predicted*, *X=None*, *sample_weight=None*)

Returns a numerical score indicating performance based on the differences between the predicted and actual values.

**Parameters**

- **y_predicted** (*pd.Series*) – Predicted values of length [n_samples]
- **y_true** (*pd.Series*) – Actual class labels of length [n_samples]
- **X** (*pd.DataFrame or np.ndarray*) – Extra data of shape [n_samples, n_features] necessary to calculate score
- **sample_weight** (*pd.DataFrame or np.ndarray*) – Sample weights used in computing objective value result

**Returns** score

**validate_inputs**(*self*, *y_true*, *y_predicted*)

Validate inputs for scoring.

**class** evalml.objectives.**AUCMacro**

AUC score for multiclass classification using macro averaging.

### Example

```
>>> y_true = [0, 1, 2, 0, 2, 1]
>>> y_pred = [[0.7, 0.2, 0.1],
...           [0.1, 0.0, 0.9],
...           [0.1, 0.3, 0.6],
...           [0.9, 0.1, 0.0],
...           [0.6, 0.1, 0.3],
...           [0.5, 0.5, 0.0]]
>>> np.testing.assert_almost_equal(AUCMacro().objective_function(y_true, y_pred), 0.
↪75)
```

**Attributes**

| | |
|---|---|
| **expected_range** | [0, 1] |
| **greater_is_better** | True |
| **is_bounded_like_percentage** | True |
| **name** | AUC Macro |
| **perfect_score** | 1.0 |
| **problem_types** | [ProblemTypes.MULTICLASS, ProblemTypes.TIME_SERIES_MULTICLASS] |
| **score_needs_proba** | True |

**Methods**

| | |
|---|---|
| *calculate_percent_difference* | Calculate the percent difference between scores. |
| *is_defined_for_problem_type* | Returns whether or not an objective is defined for a problem type. |
| *objective_function* | Objective function for AUC score for multiclass classification using macro-averaging. |
| *positive_only* | If True, this objective is only valid for positive data. Defaults to False. |
| *score* | Returns a numerical score indicating performance based on the differences between the predicted and actual values. |
| *validate_inputs* | Validates the input based on a few simple checks. |

**classmethod calculate_percent_difference**(*cls*, *score*, *baseline_score*)

Calculate the percent difference between scores.

**Parameters**

- **score** (*float*) – A score. Output of the score method of this objective.
- **baseline_score** (*float*) – A score. Output of the score method of this objective. In practice, this is the score achieved on this objective with a baseline estimator.

**Returns**

**The percent difference between the scores. Note that for objectives that can be interpreted** as percentages, this will be the difference between the reference score and score. For all other objectives, the difference will be normalized by the reference score.

**Return type** float

**classmethod is_defined_for_problem_type**(*cls*, *problem_type*)

Returns whether or not an objective is defined for a problem type.

**objective_function**(*self*, *y_true*, *y_predicted*, *X=None*, *sample_weight=None*)

Objective function for AUC score for multiclass classification using macro-averaging.

**positive_only**(*cls*)

If True, this objective is only valid for positive data. Defaults to False.

**score**(*self*, *y_true*, *y_predicted*, *X=None*, *sample_weight=None*)

Returns a numerical score indicating performance based on the differences between the predicted and actual values.

**Parameters**

- **y_predicted** (*pd.Series*) – Predicted values of length [n_samples]
- **y_true** (*pd.Series*) – Actual class labels of length [n_samples]
- **X** (*pd.DataFrame or np.ndarray*) – Extra data of shape [n_samples, n_features] necessary to calculate score
- **sample_weight** (*pd.DataFrame or np.ndarray*) – Sample weights used in computing objective value result

**Returns** score

**validate_inputs**(*self*, *y_true*, *y_predicted*)

Validates the input based on a few simple checks.

**Parameters**

- **y_predicted** (*pd.Series, or pd.DataFrame*) – Predicted values of length [n_samples].
- **y_true** (*pd.Series*) – Actual class labels of length [n_samples].

**Raises** **ValueError** – If the inputs are malformed.

**class** evalml.objectives.**AUCMicro**

AUC score for multiclass classification using micro averaging.

### Example

```
>>> y_true = [0, 1, 2, 0, 2, 1]
>>> y_pred = [[0.7, 0.2, 0.1],
...           [0.3, 0.5, 0.2],
...           [0.1, 0.3, 0.6],
...           [0.9, 0.1, 0.0],
...           [0.3, 0.1, 0.6],
...           [0.5, 0.5, 0.0]]
>>> np.testing.assert_almost_equal(AUCMicro().objective_function(y_true, y_pred), 0.
↪9861111)
```

**Attributes**

| | |
|---|---|
| **expected_range** | [0, 1] |
| **greater_is_better** | True |
| **is_bounded_likely** | True |
| **name** | AUC Micro |
| **perfect_score** | 1.0 |
| **problem_types** | [ProblemTypes.MULTICLASS, ProblemTypes.TIME_SERIES_MULTICLASS] |
| **score_needs_proba** | True |

**Methods**

| | |
|---|---|
| *calculate_percent_difference* | Calculate the percent difference between scores. |
| *is_defined_for_problem_type* | Returns whether or not an objective is defined for a problem type. |
| *objective_function* | Objective function for AUC score for multiclass classification using micro-averaging. |
| *positive_only* | If True, this objective is only valid for positive data. Defaults to False. |
| *score* | Returns a numerical score indicating performance based on the differences between the predicted and actual values. |
| *validate_inputs* | Validates the input based on a few simple checks. |

**classmethod calculate_percent_difference**(*cls*, *score*, *baseline_score*)

Calculate the percent difference between scores.

**Parameters**

- **score** (*float*) – A score. Output of the score method of this objective.
- **baseline_score** (*float*) – A score. Output of the score method of this objective. In practice, this is the score achieved on this objective with a baseline estimator.

**Returns**

**The percent difference between the scores. Note that for objectives that can be interpreted** as percentages, this will be the difference between the reference score and score. For all other objectives, the difference will be normalized by the reference score.

**Return type** float

**classmethod is_defined_for_problem_type**(*cls*, *problem_type*)

Returns whether or not an objective is defined for a problem type.

**objective_function**(*self*, *y_true*, *y_predicted*, *X=None*, *sample_weight=None*)

Objective function for AUC score for multiclass classification using micro-averaging.

**positive_only**(*cls*)

If True, this objective is only valid for positive data. Defaults to False.

**score**(*self*, *y_true*, *y_predicted*, *X=None*, *sample_weight=None*)

Returns a numerical score indicating performance based on the differences between the predicted and actual values.

**Parameters**

- **y_predicted** (*pd.Series*) – Predicted values of length [n_samples]
- **y_true** (*pd.Series*) – Actual class labels of length [n_samples]
- **X** (*pd.DataFrame or np.ndarray*) – Extra data of shape [n_samples, n_features] necessary to calculate score
- **sample_weight** (*pd.DataFrame or np.ndarray*) – Sample weights used in computing objective value result

**Returns** score

**validate_inputs**(*self*, *y_true*, *y_predicted*)

Validates the input based on a few simple checks.

**Parameters**

- **y_predicted** (*pd.Series, or pd.DataFrame*) – Predicted values of length [n_samples].
- **y_true** (*pd.Series*) – Actual class labels of length [n_samples].

**Raises** **ValueError** – If the inputs are malformed.

**class** evalml.objectives.**AUCWeighted**

AUC Score for multiclass classification using weighted averaging.

### Example

```
>>> y_true = [0, 1, 2, 0, 2, 1]
>>> y_pred = [[0.7, 0.2, 0.1],
...           [0.1, 0.0, 0.9],
...           [0.1, 0.3, 0.6],
...           [0.1, 0.2, 0.7],
...           [0.6, 0.1, 0.3],
...           [0.5, 0.2, 0.3]]
>>> np.testing.assert_almost_equal(AUCWeighted().objective_function(y_true, y_pred),
↪ 0.4375)
```

**Attributes**

| | |
|---|---|
| **expected_range** | [0, 1] |
| **greater_is_better** | True |
| **is_bounded_like** | True |
| **name** | AUC Weighted |
| **perfect_score** | 1.0 |
| **problem_types** | [ProblemTypes.MULTICLASS, ProblemTypes.TIME_SERIES_MULTICLASS] |
| **score_needs_proba** | True |

**Methods**

| | |
|---|---|
| *calculate_percent_difference* | Calculate the percent difference between scores. |
| *is_defined_for_problem_type* | Returns whether or not an objective is defined for a problem type. |
| *objective_function* | Objective function for AUC Score for multiclass classification using weighted averaging. |
| *positive_only* | If True, this objective is only valid for positive data. Defaults to False. |
| *score* | Returns a numerical score indicating performance based on the differences between the predicted and actual values. |
| *validate_inputs* | Validates the input based on a few simple checks. |

**classmethod calculate_percent_difference**(*cls*, *score*, *baseline_score*)

Calculate the percent difference between scores.

**Parameters**

- **score** (*float*) – A score. Output of the score method of this objective.
- **baseline_score** (*float*) – A score. Output of the score method of this objective. In practice, this is the score achieved on this objective with a baseline estimator.

**Returns**

**The percent difference between the scores. Note that for objectives that can be interpreted** as percentages, this will be the difference between the reference score and score. For all other objectives, the difference will be normalized by the reference score.

**Return type** float

**classmethod is_defined_for_problem_type**(*cls*, *problem_type*)

Returns whether or not an objective is defined for a problem type.

**objective_function**(*self*, *y_true*, *y_predicted*, *X=None*, *sample_weight=None*)

Objective function for AUC Score for multiclass classification using weighted averaging.

**positive_only**(*cls*)

If True, this objective is only valid for positive data. Defaults to False.

**score**(*self*, *y_true*, *y_predicted*, *X=None*, *sample_weight=None*)

Returns a numerical score indicating performance based on the differences between the predicted and actual values.

**Parameters**

- **y_predicted** (*pd.Series*) – Predicted values of length [n_samples]
- **y_true** (*pd.Series*) – Actual class labels of length [n_samples]
- **X** (*pd.DataFrame or np.ndarray*) – Extra data of shape [n_samples, n_features] necessary to calculate score
- **sample_weight** (*pd.DataFrame or np.ndarray*) – Sample weights used in computing objective value result

**Returns** score

**validate_inputs**(*self*, *y_true*, *y_predicted*)

Validates the input based on a few simple checks.

**Parameters**

- **y_predicted** (*pd.Series, or pd.DataFrame*) – Predicted values of length [n_samples].
- **y_true** (*pd.Series*) – Actual class labels of length [n_samples].

**Raises** **ValueError** – If the inputs are malformed.

**class** evalml.objectives.**BalancedAccuracyBinary**

Balanced accuracy score for binary classification.

### Example

```
>>> y_true = pd.Series([0, 0, 0, 0, 0, 1, 1, 1, 0, 1, 1])
>>> y_pred = pd.Series([0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1])
>>> np.testing.assert_almost_equal(BalancedAccuracyBinary().objective_function(y_
↪true, y_pred), 0.60)
```

**Attributes**

| | |
|---|---|
| **expected_range** | [0, 1] |
| **greater_is_better** | True |
| **is_bounded_like_percentage** | True |
| **name** | Balanced Accuracy Binary |
| **perfect_score** | 1.0 |
| **problem_types** | [ProblemTypes.BINARY, ProblemTypes.TIME_SERIES_BINARY] |
| **score_needs_proba** | False |

**Methods**

| | |
|---|---|
| *calculate_percent_difference* | Calculate the percent difference between scores. |
| *can_optimize_threshold* | Returns a boolean determining if we can optimize the binary classification objective threshold. |
| *decision_function* | Apply a learned threshold to predicted probabilities to get predicted classes. |
| *is_defined_for_problem_type* | Returns whether or not an objective is defined for a problem type. |
| *objective_function* | Objective function for accuracy score for balanced accuracy for binary classification. |
| *optimize_threshold* | Learn a binary classification threshold which optimizes the current objective. |
| *positive_only* | If True, this objective is only valid for positive data. Defaults to False. |
| *score* | Returns a numerical score indicating performance based on the differences between the predicted and actual values. |
| *validate_inputs* | Validate inputs for scoring. |

**classmethod calculate_percent_difference**(*cls*, *score*, *baseline_score*)

Calculate the percent difference between scores.

**Parameters**

- **score** (*float*) – A score. Output of the score method of this objective.
- **baseline_score** (*float*) – A score. Output of the score method of this objective. In practice, this is the score achieved on this objective with a baseline estimator.

**Returns**

**The percent difference between the scores. Note that for objectives that can be interpreted** as percentages, this will be the difference between the reference score and score. For all other objectives, the difference will be normalized by the reference score.

**Return type** float

**property can_optimize_threshold**(*cls*)

Returns a boolean determining if we can optimize the binary classification objective threshold.

This will be false for any objective that works directly with predicted probabilities, like log loss and AUC. Otherwise, it will be true.

**Returns** Whether or not an objective can be optimized.

**Return type** bool

**decision_function**(*self*, *ypred_proba*, *threshold=0.5*, *X=None*)

Apply a learned threshold to predicted probabilities to get predicted classes.

**Parameters**

- **ypred_proba** (*pd.Series, np.ndarray*) – The classifier's predicted probabilities
- **threshold** (*float, optional*) – Threshold used to make a prediction. Defaults to 0.5.
- **X** (*pd.DataFrame, optional*) – Any extra columns that are needed from training data.

**Returns** predictions

**classmethod is_defined_for_problem_type**(*cls*, *problem_type*)

Returns whether or not an objective is defined for a problem type.

**objective_function**(*self*, *y_true*, *y_predicted*, *X=None*, *sample_weight=None*)

Objective function for accuracy score for balanced accuracy for binary classification.

**optimize_threshold**(*self*, *ypred_proba*, *y_true*, *X=None*)

Learn a binary classification threshold which optimizes the current objective.

**Parameters**

- **ypred_proba** (*pd.Series*) – The classifier's predicted probabilities
- **y_true** (*pd.Series*) – The ground truth for the predictions.
- **X** (*pd.DataFrame, optional*) – Any extra columns that are needed from training data.

**Returns** Optimal threshold for this objective.

**Raises** **RuntimeError** – If objective cannot be optimized.

**positive_only**(*cls*)

If True, this objective is only valid for positive data. Defaults to False.

**score**(*self*, *y_true*, *y_predicted*, *X=None*, *sample_weight=None*)

Returns a numerical score indicating performance based on the differences between the predicted and actual values.

**Parameters**

- **y_predicted** (*pd.Series*) – Predicted values of length [n_samples]
- **y_true** (*pd.Series*) – Actual class labels of length [n_samples]
- **X** (*pd.DataFrame or np.ndarray*) – Extra data of shape [n_samples, n_features] necessary to calculate score
- **sample_weight** (*pd.DataFrame or np.ndarray*) – Sample weights used in computing objective value result

**Returns** score

**validate_inputs**(*self*, *y_true*, *y_predicted*)

Validate inputs for scoring.

**class** evalml.objectives.**BalancedAccuracyMulticlass**

Balanced accuracy score for multiclass classification.

### Example

```
>>> y_true = pd.Series([0, 1, 0, 2, 0, 1, 2, 1, 2, 0, 2])
>>> y_pred = pd.Series([0, 0, 0, 0, 1, 1, 1, 1, 2, 2, 2])
>>> np.testing.assert_almost_equal(BalancedAccuracyMulticlass().objective_
↪function(y_true, y_pred), 0.5555555)
```

**Attributes**

| | |
|---|---|
| **expected_range** | [0, 1] |
| **greater_is_better** | True |
| **is_bounded_like** | True |
| **name** | Balanced Accuracy Multiclass |
| **perfect_score** | 1.0 |
| **problem_types** | [ProblemTypes.MULTICLASS, ProblemTypes.TIME_SERIES_MULTICLASS] |
| **score_needs_proba** | False |

**Methods**

| | |
|---|---|
| *calculate_percent_difference* | Calculate the percent difference between scores. |
| *is_defined_for_problem_type* | Returns whether or not an objective is defined for a problem type. |
| *objective_function* | Objective function for accuracy score for balanced accuracy for multiclass classification. |
| *positive_only* | If True, this objective is only valid for positive data. Defaults to False. |
| *score* | Returns a numerical score indicating performance based on the differences between the predicted and actual values. |
| *validate_inputs* | Validates the input based on a few simple checks. |

**classmethod calculate_percent_difference**(*cls*, *score*, *baseline_score*)

Calculate the percent difference between scores.

**Parameters**

- **score** (*float*) – A score. Output of the score method of this objective.
- **baseline_score** (*float*) – A score. Output of the score method of this objective. In practice, this is the score achieved on this objective with a baseline estimator.

**Returns**

**The percent difference between the scores. Note that for objectives that can be interpreted** as percentages, this will be the difference between the reference score and score. For all other objectives, the difference will be normalized by the reference score.

**Return type** float

`classmethod is_defined_for_problem_type`(*cls*, *problem_type*)

Returns whether or not an objective is defined for a problem type.

`objective_function`(*self*, *y_true*, *y_predicted*, *X=None*, *sample_weight=None*)

Objective function for accuracy score for balanced accuracy for multiclass classification.

`positive_only`(*cls*)

If True, this objective is only valid for positive data. Defaults to False.

`score`(*self*, *y_true*, *y_predicted*, *X=None*, *sample_weight=None*)

Returns a numerical score indicating performance based on the differences between the predicted and actual values.

**Parameters**

- **y_predicted** (*pd.Series*) – Predicted values of length [n_samples]
- **y_true** (*pd.Series*) – Actual class labels of length [n_samples]
- **X** (*pd.DataFrame or np.ndarray*) – Extra data of shape [n_samples, n_features] necessary to calculate score
- **sample_weight** (*pd.DataFrame or np.ndarray*) – Sample weights used in computing objective value result

**Returns** score

`validate_inputs`(*self*, *y_true*, *y_predicted*)

Validates the input based on a few simple checks.

**Parameters**

- **y_predicted** (*pd.Series, or pd.DataFrame*) – Predicted values of length [n_samples].
- **y_true** (*pd.Series*) – Actual class labels of length [n_samples].

**Raises** **ValueError** – If the inputs are malformed.

`class evalml.objectives.BinaryClassificationObjective`

Base class for all binary classification objectives.

**Attributes**

| | |
|---|---|
| **problem_types** | [ProblemTypes.BINARY, ProblemTypes.TIME_SERIES_BINARY] |

**Methods**

| | |
|---|---|
| *calculate_percent_difference* | Calculate the percent difference between scores. |
| *can_optimize_threshold* | Returns a boolean determining if we can optimize the binary classification objective threshold. |
| *decision_function* | Apply a learned threshold to predicted probabilities to get predicted classes. |
| *expected_range* | Returns the expected range of the objective, which is not necessarily the possible ranges. |
| *greater_is_better* | Returns a boolean determining if a greater score indicates better model performance. |
| *is_bounded_like_percentage* | Returns whether this objective is bounded between 0 and 1, inclusive. |
| *is_defined_for_problem_type* | Returns whether or not an objective is defined for a problem type. |
| *name* | Returns a name describing the objective. |
| *objective_function* | Computes the relative value of the provided predictions compared to the actual labels, according a specified metric. |
| *optimize_threshold* | Learn a binary classification threshold which optimizes the current objective. |
| *perfect_score* | Returns the score obtained by evaluating this objective on a perfect model. |
| *positive_only* | If True, this objective is only valid for positive data. Defaults to False. |
| *score* | Returns a numerical score indicating performance based on the differences between the predicted and actual values. |
| *score_needs_proba* | Returns a boolean determining if the score() method needs probability estimates. |
| *validate_inputs* | Validate inputs for scoring. |

**classmethod calculate_percent_difference**(*cls*, *score*, *baseline_score*)

Calculate the percent difference between scores.

**Parameters**

- **score** (*float*) – A score. Output of the score method of this objective.
- **baseline_score** (*float*) – A score. Output of the score method of this objective. In practice, this is the score achieved on this objective with a baseline estimator.

**Returns**

**The percent difference between the scores. Note that for objectives that can be interpreted** as percentages, this will be the difference between the reference score and score. For all other objectives, the difference will be normalized by the reference score.

**Return type** float

**property can_optimize_threshold**(*cls*)

Returns a boolean determining if we can optimize the binary classification objective threshold.

This will be false for any objective that works directly with predicted probabilities, like log loss and AUC. Otherwise, it will be true.

**Returns** Whether or not an objective can be optimized.

**Return type** bool

**decision_function**(*self*, *ypred_proba*, *threshold=0.5*, *X=None*)

Apply a learned threshold to predicted probabilities to get predicted classes.

**Parameters**

- **ypred_proba** (*pd.Series, np.ndarray*) – The classifier's predicted probabilities
- **threshold** (*float, optional*) – Threshold used to make a prediction. Defaults to 0.5.
- **X** (*pd.DataFrame, optional*) – Any extra columns that are needed from training data.

**Returns** predictions

**property expected_range**(*cls*)

Returns the expected range of the objective, which is not necessarily the possible ranges.

For example, our expected R2 range is from [-1, 1], although the actual range is (-inf, 1].

**property greater_is_better**(*cls*)

Returns a boolean determining if a greater score indicates better model performance.

**property is_bounded_like_percentage**(*cls*)

Returns whether this objective is bounded between 0 and 1, inclusive.

**classmethod is_defined_for_problem_type**(*cls*, *problem_type*)

Returns whether or not an objective is defined for a problem type.

**property name**(*cls*)

Returns a name describing the objective.

**abstract classmethod objective_function**(*cls*, *y_true*, *y_predicted*, *X=None*, *sample_weight=None*)

Computes the relative value of the provided predictions compared to the actual labels, according a specified metric.

**Parameters**

- **y_predicted** (*pd.Series*) – Predicted values of length [n_samples]
- **y_true** (*pd.Series*) – Actual class labels of length [n_samples]
- **X** (*pd.DataFrame or np.ndarray*) – Extra data of shape [n_samples, n_features] necessary to calculate score
- **sample_weight** (*pd.DataFrame or np.ndarray*) – Sample weights used in computing objective value result

**Returns** Numerical value used to calculate score

**optimize_threshold**(*self*, *ypred_proba*, *y_true*, *X=None*)

Learn a binary classification threshold which optimizes the current objective.

**Parameters**

- **ypred_proba** (*pd.Series*) – The classifier's predicted probabilities
- **y_true** (*pd.Series*) – The ground truth for the predictions.
- **X** (*pd.DataFrame, optional*) – Any extra columns that are needed from training data.

**Returns** Optimal threshold for this objective.

**Raises** **RuntimeError** – If objective cannot be optimized.

**property perfect_score**(*cls*)

Returns the score obtained by evaluating this objective on a perfect model.

**positive_only**(*cls*)

If True, this objective is only valid for positive data. Defaults to False.

**score**(*self*, *y_true*, *y_predicted*, *X=None*, *sample_weight=None*)

Returns a numerical score indicating performance based on the differences between the predicted and actual values.

**Parameters**

- **y_predicted** (*pd.Series*) – Predicted values of length [n_samples]
- **y_true** (*pd.Series*) – Actual class labels of length [n_samples]
- **X** (*pd.DataFrame or np.ndarray*) – Extra data of shape [n_samples, n_features] necessary to calculate score
- **sample_weight** (*pd.DataFrame or np.ndarray*) – Sample weights used in computing objective value result

**Returns** score

**property score_needs_proba**(*cls*)

Returns a boolean determining if the score() method needs probability estimates.

This should be true for objectives which work with predicted probabilities, like log loss or AUC, and false for objectives which compare predicted class labels to the actual labels, like F1 or correlation.

**validate_inputs**(*self*, *y_true*, *y_predicted*)

Validate inputs for scoring.

**class** evalml.objectives.**CostBenefitMatrix**(*true_positive*, *true_negative*, *false_positive*, *false_negative*)

Score using a cost-benefit matrix. Scores quantify the benefits of a given value, so greater numeric scores represents a better score. Costs and scores can be negative, indicating that a value is not beneficial. For example, in the case of monetary profit, a negative cost and/or score represents loss of cash flow.

**Parameters**

- **true_positive** (*float*) – Cost associated with true positive predictions.
- **true_negative** (*float*) – Cost associated with true negative predictions.
- **false_positive** (*float*) – Cost associated with false positive predictions.
- **false_negative** (*float*) – Cost associated with false negative predictions.

**Attributes**

| | |
|---|---|
| **expected_range** | None |
| **greater_is_better** | True |
| **is_bounded_like_percentage** | False |
| **name** | Cost Benefit Matrix |
| **perfect_score** | None |
| **problem_types** | [ProblemTypes.BINARY, ProblemTypes.TIME_SERIES_BINARY] |
| **score_needs_proba** | False |

**Methods**

| | |
|---|---|
| *calculate_percent_difference* | Calculate the percent difference between scores. |
| *can_optimize_threshold* | Returns a boolean determining if we can optimize the binary classification objective threshold. |
| *decision_function* | Apply a learned threshold to predicted probabilities to get predicted classes. |
| *is_defined_for_problem_type* | Returns whether or not an objective is defined for a problem type. |
| *objective_function* | Calculates cost-benefit of the using the predicted and true values. |
| *optimize_threshold* | Learn a binary classification threshold which optimizes the current objective. |
| *positive_only* | If True, this objective is only valid for positive data. Defaults to False. |
| *score* | Returns a numerical score indicating performance based on the differences between the predicted and actual values. |
| *validate_inputs* | Validate inputs for scoring. |

**classmethod calculate_percent_difference**(*cls*, *score*, *baseline_score*)

Calculate the percent difference between scores.

**Parameters**

- **score** (*float*) – A score. Output of the score method of this objective.
- **baseline_score** (*float*) – A score. Output of the score method of this objective. In practice, this is the score achieved on this objective with a baseline estimator.

**Returns**

**The percent difference between the scores. Note that for objectives that can be interpreted** as percentages, this will be the difference between the reference score and score. For all other objectives, the difference will be normalized by the reference score.

**Return type** float

**property can_optimize_threshold**(*cls*)

Returns a boolean determining if we can optimize the binary classification objective threshold.

This will be false for any objective that works directly with predicted probabilities, like log loss and AUC. Otherwise, it will be true.

**Returns** Whether or not an objective can be optimized.

**Return type** bool

**decision_function**(*self*, *ypred_proba*, *threshold=0.5*, *X=None*)

Apply a learned threshold to predicted probabilities to get predicted classes.

**Parameters**

- **ypred_proba** (*pd.Series, np.ndarray*) – The classifier's predicted probabilities
- **threshold** (*float, optional*) – Threshold used to make a prediction. Defaults to 0.5.
- **X** (*pd.DataFrame, optional*) – Any extra columns that are needed from training data.

**Returns** predictions

**classmethod is_defined_for_problem_type**(*cls*, *problem_type*)

Returns whether or not an objective is defined for a problem type.

**objective_function**(*self*, *y_true*, *y_predicted*, *X=None*, *sample_weight=None*)

Calculates cost-benefit of the using the predicted and true values.

**Parameters**

- **y_predicted** (*pd.Series*) – Predicted labels.
- **y_true** (*pd.Series*) – True labels.
- **X** (*pd.DataFrame*) – Ignored.
- **sample_weight** (*pd.DataFrame*) – Ignored.

**Returns** Cost-benefit matrix score

**Return type** float

**optimize_threshold**(*self*, *ypred_proba*, *y_true*, *X=None*)

Learn a binary classification threshold which optimizes the current objective.

**Parameters**

- **ypred_proba** (*pd.Series*) – The classifier's predicted probabilities
- **y_true** (*pd.Series*) – The ground truth for the predictions.
- **X** (*pd.DataFrame, optional*) – Any extra columns that are needed from training data.

**Returns** Optimal threshold for this objective.

**Raises** **RuntimeError** – If objective cannot be optimized.

**positive_only**(*cls*)

If True, this objective is only valid for positive data. Defaults to False.

**score**(*self*, *y_true*, *y_predicted*, *X=None*, *sample_weight=None*)

Returns a numerical score indicating performance based on the differences between the predicted and actual values.

**Parameters**

- **y_predicted** (*pd.Series*) – Predicted values of length [n_samples]
- **y_true** (*pd.Series*) – Actual class labels of length [n_samples]
- **X** (*pd.DataFrame or np.ndarray*) – Extra data of shape [n_samples, n_features] necessary to calculate score
- **sample_weight** (*pd.DataFrame or np.ndarray*) – Sample weights used in computing objective value result

**Returns** score

**validate_inputs**(*self*, *y_true*, *y_predicted*)

Validate inputs for scoring.

**class** evalml.objectives.**ExpVariance**

Explained variance score for regression.

### Example

```
>>> y_true = pd.Series([1.5, 2, 3, 1, 0.5, 1, 2.5, 2.5, 1, 0.5, 2])
>>> y_pred = pd.Series([1.5, 2.5, 2, 1, 0.5, 1, 3, 2.25, 0.75, 0.25, 1.75])
>>> np.testing.assert_almost_equal(ExpVariance().objective_function(y_true, y_pred),
↪ 0.7760736)
```

**Attributes**

| | |
|---|---|
| **expected_range** | None |
| **greater_is_better** | True |
| **is_bounded_like_percentage** | False |
| **name** | ExpVariance |
| **perfect_score** | 1.0 |
| **problem_types** | [ProblemTypes.REGRESSION, ProblemTypes.TIME_SERIES_REGRESSION] |
| **score_needs_proba** | False |

**Methods**

| | |
|---|---|
| *calculate_percent_difference* | Calculate the percent difference between scores. |
| *is_defined_for_problem_type* | Returns whether or not an objective is defined for a problem type. |
| *objective_function* | Objective function for explained variance score for regression. |
| *positive_only* | If True, this objective is only valid for positive data. Defaults to False. |
| *score* | Returns a numerical score indicating performance based on the differences between the predicted and actual values. |
| *validate_inputs* | Validates the input based on a few simple checks. |

**classmethod calculate_percent_difference**(*cls*, *score*, *baseline_score*)

Calculate the percent difference between scores.

**Parameters**

- **score** (*float*) – A score. Output of the score method of this objective.
- **baseline_score** (*float*) – A score. Output of the score method of this objective. In practice, this is the score achieved on this objective with a baseline estimator.

**Returns**

**The percent difference between the scores. Note that for objectives that can be interpreted** as percentages, this will be the difference between the reference score and score. For all other objectives, the difference will be normalized by the reference score.

**Return type** float

**classmethod is_defined_for_problem_type**(*cls*, *problem_type*)

Returns whether or not an objective is defined for a problem type.

**objective_function**(*self*, *y_true*, *y_predicted*, *X=None*, *sample_weight=None*)

Objective function for explained variance score for regression.

**positive_only**(*cls*)

If True, this objective is only valid for positive data. Defaults to False.

**score**(*self*, *y_true*, *y_predicted*, *X=None*, *sample_weight=None*)

Returns a numerical score indicating performance based on the differences between the predicted and actual values.

**Parameters**

- **y_predicted** (*pd.Series*) – Predicted values of length [n_samples]
- **y_true** (*pd.Series*) – Actual class labels of length [n_samples]
- **X** (*pd.DataFrame or np.ndarray*) – Extra data of shape [n_samples, n_features] necessary to calculate score
- **sample_weight** (*pd.DataFrame or np.ndarray*) – Sample weights used in computing objective value result

**Returns** score

**validate_inputs**(*self*, *y_true*, *y_predicted*)

Validates the input based on a few simple checks.

**Parameters**

- **y_predicted** (*pd.Series, or pd.DataFrame*) – Predicted values of length [n_samples].
- **y_true** (*pd.Series*) – Actual class labels of length [n_samples].

**Raises** **ValueError** – If the inputs are malformed.

**class** evalml.objectives.**F1**

F1 score for binary classification.

### Example

```
>>> y_true = pd.Series([0, 1, 1, 0, 0, 1, 1, 1, 0, 1, 1])
>>> y_pred = pd.Series([0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1])
>>> np.testing.assert_almost_equal(F1().objective_function(y_true, y_pred), 0.25)
```

**Attributes**

| | |
|---|---|
| **expected_range** | [0, 1] |
| **greater_is_better** | True |
| **is_bounded_like_percentage** | True |
| **name** | F1 |
| **perfect_score** | 1.0 |
| **problem_types** | [ProblemTypes.BINARY, ProblemTypes.TIME_SERIES_BINARY] |
| **score_needs_proba** | False |

**Methods**

| | |
|---|---|
| *calculate_percent_difference* | Calculate the percent difference between scores. |
| *can_optimize_threshold* | Returns a boolean determining if we can optimize the binary classification objective threshold. |
| *decision_function* | Apply a learned threshold to predicted probabilities to get predicted classes. |
| *is_defined_for_problem_type* | Returns whether or not an objective is defined for a problem type. |
| *objective_function* | Objective function for F1 score for binary classification. |
| *optimize_threshold* | Learn a binary classification threshold which optimizes the current objective. |
| *positive_only* | If True, this objective is only valid for positive data. Defaults to False. |
| *score* | Returns a numerical score indicating performance based on the differences between the predicted and actual values. |
| *validate_inputs* | Validate inputs for scoring. |

**classmethod calculate_percent_difference**(*cls*, *score*, *baseline_score*)

Calculate the percent difference between scores.

**Parameters**

- **score** (*float*) – A score. Output of the score method of this objective.
- **baseline_score** (*float*) – A score. Output of the score method of this objective. In practice, this is the score achieved on this objective with a baseline estimator.

**Returns**

**The percent difference between the scores. Note that for objectives that can be interpreted** as percentages, this will be the difference between the reference score and score. For all other objectives, the difference will be normalized by the reference score.

**Return type** float

**property can_optimize_threshold**(*cls*)

Returns a boolean determining if we can optimize the binary classification objective threshold.

This will be false for any objective that works directly with predicted probabilities, like log loss and AUC. Otherwise, it will be true.

**Returns** Whether or not an objective can be optimized.

**Return type** bool

**decision_function**(*self*, *ypred_proba*, *threshold=0.5*, *X=None*)

Apply a learned threshold to predicted probabilities to get predicted classes.

**Parameters**

- **ypred_proba** (*pd.Series, np.ndarray*) – The classifier's predicted probabilities
- **threshold** (*float, optional*) – Threshold used to make a prediction. Defaults to 0.5.
- **X** (*pd.DataFrame, optional*) – Any extra columns that are needed from training data.

**Returns** predictions

**classmethod is_defined_for_problem_type**(*cls*, *problem_type*)

Returns whether or not an objective is defined for a problem type.

**objective_function**(*self*, *y_true*, *y_predicted*, *X=None*, *sample_weight=None*)

Objective function for F1 score for binary classification.

**optimize_threshold**(*self*, *ypred_proba*, *y_true*, *X=None*)

Learn a binary classification threshold which optimizes the current objective.

**Parameters**

- **ypred_proba** (*pd.Series*) – The classifier's predicted probabilities
- **y_true** (*pd.Series*) – The ground truth for the predictions.
- **X** (*pd.DataFrame, optional*) – Any extra columns that are needed from training data.

**Returns** Optimal threshold for this objective.

**Raises** **RuntimeError** – If objective cannot be optimized.

**positive_only**(*cls*)

If True, this objective is only valid for positive data. Defaults to False.

**score**(*self*, *y_true*, *y_predicted*, *X=None*, *sample_weight=None*)

Returns a numerical score indicating performance based on the differences between the predicted and actual values.

**Parameters**

- **y_predicted** (*pd.Series*) – Predicted values of length [n_samples]
- **y_true** (*pd.Series*) – Actual class labels of length [n_samples]
- **X** (*pd.DataFrame or np.ndarray*) – Extra data of shape [n_samples, n_features] necessary to calculate score
- **sample_weight** (*pd.DataFrame or np.ndarray*) – Sample weights used in computing objective value result

**Returns** score

**validate_inputs**(*self*, *y_true*, *y_predicted*)

Validate inputs for scoring.

**class** evalml.objectives.**F1Macro**

F1 score for multiclass classification using macro averaging.

### Example

```
>>> y_true = pd.Series([0, 1, 0, 2, 0, 1, 2, 1, 2, 0, 2])
>>> y_pred = pd.Series([0, 0, 0, 0, 1, 1, 1, 1, 2, 2, 2])
>>> np.testing.assert_almost_equal(F1Macro().objective_function(y_true, y_pred), 0.
↪5476190)
```

**Attributes**

| | |
|---|---|
| **expected_range** | [0, 1] |
| **greater_is_better** | True |
| **is_bounded_likely** | True |
| **name** | F1 Macro |
| **perfect_score** | 1.0 |
| **problem_types** | [ProblemTypes.MULTICLASS, ProblemTypes.TIME_SERIES_MULTICLASS] |
| **score_needs_proba** | False |

**Methods**

| | |
|---|---|
| *calculate_percent_difference* | Calculate the percent difference between scores. |
| *is_defined_for_problem_type* | Returns whether or not an objective is defined for a problem type. |
| *objective_function* | Objective function for F1 score for multiclass classification using macro averaging. |
| *positive_only* | If True, this objective is only valid for positive data. Defaults to False. |
| *score* | Returns a numerical score indicating performance based on the differences between the predicted and actual values. |
| *validate_inputs* | Validates the input based on a few simple checks. |

**classmethod calculate_percent_difference**(*cls*, *score*, *baseline_score*)

Calculate the percent difference between scores.

**Parameters**

- **score** (*float*) – A score. Output of the score method of this objective.
- **baseline_score** (*float*) – A score. Output of the score method of this objective. In practice, this is the score achieved on this objective with a baseline estimator.

**Returns**

**The percent difference between the scores. Note that for objectives that can be interpreted** as percentages, this will be the difference between the reference score and score. For all other objectives, the difference will be normalized by the reference score.

**Return type** float

**classmethod is_defined_for_problem_type**(*cls*, *problem_type*)

Returns whether or not an objective is defined for a problem type.

**objective_function**(*self*, *y_true*, *y_predicted*, *X=None*, *sample_weight=None*)

Objective function for F1 score for multiclass classification using macro averaging.

**positive_only**(*cls*)

If True, this objective is only valid for positive data. Defaults to False.

**score**(*self*, *y_true*, *y_predicted*, *X=None*, *sample_weight=None*)

Returns a numerical score indicating performance based on the differences between the predicted and actual values.

**Parameters**

- **y_predicted** (*pd.Series*) – Predicted values of length [n_samples]
- **y_true** (*pd.Series*) – Actual class labels of length [n_samples]
- **X** (*pd.DataFrame or np.ndarray*) – Extra data of shape [n_samples, n_features] necessary to calculate score
- **sample_weight** (*pd.DataFrame or np.ndarray*) – Sample weights used in computing objective value result

**Returns** score

**validate_inputs**(*self*, *y_true*, *y_predicted*)

Validates the input based on a few simple checks.

**Parameters**

- **y_predicted** (*pd.Series, or pd.DataFrame*) – Predicted values of length [n_samples].
- **y_true** (*pd.Series*) – Actual class labels of length [n_samples].

**Raises** **ValueError** – If the inputs are malformed.

**class** evalml.objectives.**F1Micro**

F1 score for multiclass classification using micro averaging.

### Example

```
>>> y_true = pd.Series([0, 1, 0, 2, 0, 1, 2, 1, 2, 0, 2])
>>> y_pred = pd.Series([0, 0, 0, 0, 1, 1, 1, 1, 2, 2, 2])
>>> np.testing.assert_almost_equal(F1Micro().objective_function(y_true, y_pred), 0.
↪5454545)
```

**Attributes**

| | |
|---|---|
| **expected_range** | [0, 1] |
| **greater_is_better** | True |
| **is_bounded_like** | True |
| **name** | F1 Micro |
| **perfect_score** | 1.0 |
| **problem_types** | [ProblemTypes.MULTICLASS, ProblemTypes.TIME_SERIES_MULTICLASS] |
| **score_needs_proba** | False |

**Methods**

| | |
|---|---|
| *calculate_percent_difference* | Calculate the percent difference between scores. |
| *is_defined_for_problem_type* | Returns whether or not an objective is defined for a problem type. |
| *objective_function* | Objective function for F1 score for multiclass classification. |
| *positive_only* | If True, this objective is only valid for positive data. Defaults to False. |
| *score* | Returns a numerical score indicating performance based on the differences between the predicted and actual values. |
| *validate_inputs* | Validates the input based on a few simple checks. |

**classmethod calculate_percent_difference**(*cls*, *score*, *baseline_score*)

Calculate the percent difference between scores.

**Parameters**

- **score** (*float*) – A score. Output of the score method of this objective.
- **baseline_score** (*float*) – A score. Output of the score method of this objective. In practice, this is the score achieved on this objective with a baseline estimator.

**Returns**

**The percent difference between the scores. Note that for objectives that can be interpreted** as percentages, this will be the difference between the reference score and score. For all other objectives, the difference will be normalized by the reference score.

**Return type** float

**classmethod is_defined_for_problem_type**(*cls*, *problem_type*)

Returns whether or not an objective is defined for a problem type.

**objective_function**(*self*, *y_true*, *y_predicted*, *X=None*, *sample_weight=None*)

Objective function for F1 score for multiclass classification.

**positive_only**(*cls*)

If True, this objective is only valid for positive data. Defaults to False.

**score**(*self*, *y_true*, *y_predicted*, *X=None*, *sample_weight=None*)

Returns a numerical score indicating performance based on the differences between the predicted and actual values.

**Parameters**

- **y_predicted** (*pd.Series*) – Predicted values of length [n_samples]
- **y_true** (*pd.Series*) – Actual class labels of length [n_samples]
- **X** (*pd.DataFrame or np.ndarray*) – Extra data of shape [n_samples, n_features] necessary to calculate score
- **sample_weight** (*pd.DataFrame or np.ndarray*) – Sample weights used in computing objective value result

**Returns** score

**validate_inputs**(*self*, *y_true*, *y_predicted*)

Validates the input based on a few simple checks.

**Parameters**

- **y_predicted** (*pd.Series, or pd.DataFrame*) – Predicted values of length [n_samples].
- **y_true** (*pd.Series*) – Actual class labels of length [n_samples].

**Raises** **ValueError** – If the inputs are malformed.

**class** evalml.objectives.**F1Weighted**

F1 score for multiclass classification using weighted averaging.

### Example

```
>>> y_true = pd.Series([0, 1, 0, 2, 0, 1, 2, 1, 2, 0, 2])
>>> y_pred = pd.Series([0, 0, 0, 0, 1, 1, 1, 1, 2, 2, 2])
>>> np.testing.assert_almost_equal(F1Weighted().objective_function(y_true, y_pred),␣
↪0.5454545)
```

**Attributes**

| | |
|---|---|
| **expected_range** | [0, 1] |
| **greater_is_better** | True |
| **is_bounded_like** | True |
| **name** | F1 Weighted |
| **perfect_score** | 1.0 |
| **problem_types** | [ProblemTypes.MULTICLASS, ProblemTypes.TIME_SERIES_MULTICLASS] |
| **score_needs_proba** | False |

**Methods**

| | |
|---|---|
| *calculate_percent_difference* | Calculate the percent difference between scores. |
| *is_defined_for_problem_type* | Returns whether or not an objective is defined for a problem type. |
| *objective_function* | Objective function for F1 score for multiclass classification using weighted averaging. |
| *positive_only* | If True, this objective is only valid for positive data. Defaults to False. |
| *score* | Returns a numerical score indicating performance based on the differences between the predicted and actual values. |
| *validate_inputs* | Validates the input based on a few simple checks. |

**classmethod calculate_percent_difference**(*cls*, *score*, *baseline_score*)

Calculate the percent difference between scores.

**Parameters**

- **score** (*float*) – A score. Output of the score method of this objective.
- **baseline_score** (*float*) – A score. Output of the score method of this objective. In practice, this is the score achieved on this objective with a baseline estimator.

**Returns**

**The percent difference between the scores. Note that for objectives that can be interpreted** as percentages, this will be the difference between the reference score and score. For all other objectives, the difference will be normalized by the reference score.

**Return type** float

`classmethod is_defined_for_problem_type`(*cls*, *problem_type*)

Returns whether or not an objective is defined for a problem type.

`objective_function`(*self*, *y_true*, *y_predicted*, *X=None*, *sample_weight=None*)

Objective function for F1 score for multiclass classification using weighted averaging.

`positive_only`(*cls*)

If True, this objective is only valid for positive data. Defaults to False.

`score`(*self*, *y_true*, *y_predicted*, *X=None*, *sample_weight=None*)

Returns a numerical score indicating performance based on the differences between the predicted and actual values.

**Parameters**

- **y_predicted** (*pd.Series*) – Predicted values of length [n_samples]
- **y_true** (*pd.Series*) – Actual class labels of length [n_samples]
- **X** (*pd.DataFrame or np.ndarray*) – Extra data of shape [n_samples, n_features] necessary to calculate score
- **sample_weight** (*pd.DataFrame or np.ndarray*) – Sample weights used in computing objective value result

**Returns** score

`validate_inputs`(*self*, *y_true*, *y_predicted*)

Validates the input based on a few simple checks.

**Parameters**

- **y_predicted** (*pd.Series, or pd.DataFrame*) – Predicted values of length [n_samples].
- **y_true** (*pd.Series*) – Actual class labels of length [n_samples].

**Raises** **ValueError** – If the inputs are malformed.

`class evalml.objectives.FraudCost`(*retry_percentage=0.5*, *interchange_fee=0.02*, *fraud_payout_percentage=1.0*, *amount_col='amount'*)

Score the percentage of money lost of the total transaction amount process due to fraud.

**Parameters**

- **retry_percentage** (*float*) – What percentage of customers that will retry a transaction if it is declined. Between 0 and 1. Defaults to 0.5.
- **interchange_fee** (*float*) – How much of each successful transaction you pay. Between 0 and 1. Defaults to 0.02.
- **fraud_payout_percentage** (*float*) – Percentage of fraud you will not be able to collect. Between 0 and 1. Defaults to 1.0.
- **amount_col** (*str*) – Name of column in data that contains the amount. Defaults to "amount".

**Attributes**

| | |
|---|---|
| **expected_range** | None |
| **greater_is_better** | False |
| **is_bounded_like_percentage** | True |
| **name** | Fraud Cost |
| **perfect_score** | 0.0 |
| **problem_types** | [ProblemTypes.BINARY, ProblemTypes.TIME_SERIES_BINARY] |
| **score_needs_proba** | False |

**Methods**

| | |
|---|---|
| *calculate_percent_difference* | Calculate the percent difference between scores. |
| *can_optimize_threshold* | Returns a boolean determining if we can optimize the binary classification objective threshold. |
| *decision_function* | Apply a learned threshold to predicted probabilities to get predicted classes. |
| *is_defined_for_problem_type* | Returns whether or not an objective is defined for a problem type. |
| *objective_function* | Calculate amount lost to fraud per transaction given predictions, true values, and dataframe with transaction amount. |
| *optimize_threshold* | Learn a binary classification threshold which optimizes the current objective. |
| *positive_only* | If True, this objective is only valid for positive data. Defaults to False. |
| *score* | Returns a numerical score indicating performance based on the differences between the predicted and actual values. |
| *validate_inputs* | Validate inputs for scoring. |

**classmethod calculate_percent_difference**(*cls*, *score*, *baseline_score*)

Calculate the percent difference between scores.

**Parameters**

- **score** (*float*) – A score. Output of the score method of this objective.
- **baseline_score** (*float*) – A score. Output of the score method of this objective. In practice, this is the score achieved on this objective with a baseline estimator.

**Returns**

**The percent difference between the scores. Note that for objectives that can be interpreted** as percentages, this will be the difference between the reference score and score. For all other objectives, the difference will be normalized by the reference score.

**Return type** float

**property can_optimize_threshold**(*cls*)

Returns a boolean determining if we can optimize the binary classification objective threshold.

This will be false for any objective that works directly with predicted probabilities, like log loss and AUC. Otherwise, it will be true.

**Returns** Whether or not an objective can be optimized.

**Return type** bool

`decision_function`(*self*, *ypred_proba*, *threshold=0.5*, *X=None*)

Apply a learned threshold to predicted probabilities to get predicted classes.

**Parameters**

- **ypred_proba** (*pd.Series, np.ndarray*) – The classifier's predicted probabilities
- **threshold** (*float, optional*) – Threshold used to make a prediction. Defaults to 0.5.
- **X** (*pd.DataFrame, optional*) – Any extra columns that are needed from training data.

**Returns** predictions

`classmethod is_defined_for_problem_type`(*cls*, *problem_type*)

Returns whether or not an objective is defined for a problem type.

`objective_function`(*self*, *y_true*, *y_predicted*, *X*, *sample_weight=None*)

Calculate amount lost to fraud per transaction given predictions, true values, and dataframe with transaction amount.

**Parameters**

- **y_predicted** (*pd.Series*) – Predicted fraud labels.
- **y_true** (*pd.Series*) – True fraud labels.
- **X** (*pd.DataFrame*) – Data with transaction amounts.
- **sample_weight** (*pd.DataFrame*) – Ignored.

**Returns** Amount lost to fraud per transaction.

**Return type** float

**Raises** **ValueError** – If amount_col is not a valid column in the input data.

`optimize_threshold`(*self*, *ypred_proba*, *y_true*, *X=None*)

Learn a binary classification threshold which optimizes the current objective.

**Parameters**

- **ypred_proba** (*pd.Series*) – The classifier's predicted probabilities
- **y_true** (*pd.Series*) – The ground truth for the predictions.
- **X** (*pd.DataFrame, optional*) – Any extra columns that are needed from training data.

**Returns** Optimal threshold for this objective.

**Raises** **RuntimeError** – If objective cannot be optimized.

`positive_only`(*cls*)

If True, this objective is only valid for positive data. Defaults to False.

`score`(*self*, *y_true*, *y_predicted*, *X=None*, *sample_weight=None*)

Returns a numerical score indicating performance based on the differences between the predicted and actual values.

**Parameters**

- **y_predicted** (*pd.Series*) – Predicted values of length [n_samples]
- **y_true** (*pd.Series*) – Actual class labels of length [n_samples]
- **X** (*pd.DataFrame or np.ndarray*) – Extra data of shape [n_samples, n_features] necessary to calculate score
- **sample_weight** (*pd.DataFrame or np.ndarray*) – Sample weights used in computing objective value result

**Returns** score

**validate_inputs**(*self*, *y_true*, *y_predicted*)

Validate inputs for scoring.

evalml.objectives.**get_all_objective_names**()

Get a list of the names of all objectives.

**Returns** Objective names

**Return type** list (str)

evalml.objectives.**get_core_objective_names**()

Get a list of all valid core objectives.

**Returns** Objective names.

**Return type** list[str]

evalml.objectives.**get_core_objectives**(*problem_type*)

Returns all core objective instances associated with the given problem type.

Core objectives are designed to work out-of-the-box for any dataset.

**Parameters** **problem_type** (*str/ProblemTypes*) – Type of problem

**Returns** List of ObjectiveBase instances

### Examples

```
>>> for objective in get_core_objectives("regression"):
...     print(objective.name)
ExpVariance
MaxError
MedianAE
MSE
MAE
R2
Root Mean Squared Error
>>> for objective in get_core_objectives("binary"):
...     print(objective.name)
MCC Binary
Log Loss Binary
Gini
AUC
Precision
F1
Balanced Accuracy Binary
Accuracy Binary
```

`evalml.objectives.get_non_core_objectives()`

Get non-core objective classes.

Non-core objectives are objectives that are domain-specific. Users typically need to configure these objectives before using them in AutoMLSearch.

**Returns** List of ObjectiveBase classes

`evalml.objectives.get_objective(objective, return_instance=False, **kwargs)`

Returns the Objective class corresponding to a given objective name.

**Parameters**

- **objective** (*str or ObjectiveBase*) – Name or instance of the objective class.
- **return_instance** (*bool*) – Whether to return an instance of the objective. This only applies if objective is of type str. Note that the instance will be initialized with default arguments.
- **kwargs** (*Any*) – Any keyword arguments to pass into the objective. Only used when return_instance=True.

**Returns** ObjectiveBase if the parameter objective is of type ObjectiveBase. If objective is instead a valid objective name, function will return the class corresponding to that name. If return_instance is True, an instance of that objective will be returned.

**Raises**

- **TypeError** – If objective is None.
- **TypeError** – If objective is not a string and not an instance of ObjectiveBase.
- **ObjectiveNotFoundError** – If input objective is not a valid objective.
- **ObjectiveCreationError** – If objective cannot be created properly.

`evalml.objectives.get_optimization_objectives(problem_type)`

Get objectives for optimization.

**Parameters** **problem_type** (*str/ProblemTypes*) – Type of problem

**Returns** List of ObjectiveBase instances

`evalml.objectives.get_ranking_objectives(problem_type)`

Get objectives for pipeline rankings.

**Parameters** **problem_type** (*str/ProblemTypes*) – Type of problem

**Returns** List of ObjectiveBase instances

`class evalml.objectives.Gini`

Gini coefficient for binary classification.

### Example

```
>>> y_true = pd.Series([0, 1, 1, 0, 0, 1, 1, 1, 0, 1, 1])
>>> y_pred = pd.Series([0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1])
>>> np.testing.assert_almost_equal(Gini().objective_function(y_true, y_pred), 0.
↪1428571)
```

**Attributes**

| | |
|---|---|
| **expected_range** | None |
| **greater_is_better** | True |
| **is_bounded_like_a_percentage** | False |
| **name** | Gini |
| **perfect_score** | 1.0 |
| **problem_types** | [ProblemTypes.BINARY, ProblemTypes.TIME_SERIES_BINARY] |
| **score_needs_proba** | True |

**Methods**

| | |
|---|---|
| *calculate_percent_difference* | Calculate the percent difference between scores. |
| *can_optimize_threshold* | Returns a boolean determining if we can optimize the binary classification objective threshold. |
| *decision_function* | Apply a learned threshold to predicted probabilities to get predicted classes. |
| *is_defined_for_problem_type* | Returns whether or not an objective is defined for a problem type. |
| *objective_function* | Objective function for Gini coefficient for binary classification. |
| *optimize_threshold* | Learn a binary classification threshold which optimizes the current objective. |
| *positive_only* | If True, this objective is only valid for positive data. Defaults to False. |
| *score* | Returns a numerical score indicating performance based on the differences between the predicted and actual values. |
| *validate_inputs* | Validate inputs for scoring. |

**classmethod calculate_percent_difference**(*cls*, *score*, *baseline_score*)

Calculate the percent difference between scores.

**Parameters**

- **score** (*float*) – A score. Output of the score method of this objective.
- **baseline_score** (*float*) – A score. Output of the score method of this objective. In practice, this is the score achieved on this objective with a baseline estimator.

**Returns**

**The percent difference between the scores. Note that for objectives that can be interpreted** as percentages, this will be the difference between the reference score and score. For all other objectives, the difference will be normalized by the reference score.

**Return type** float

**property can_optimize_threshold**(*cls*)

Returns a boolean determining if we can optimize the binary classification objective threshold.

This will be false for any objective that works directly with predicted probabilities, like log loss and AUC. Otherwise, it will be true.

**Returns** Whether or not an objective can be optimized.

**Return type** bool

**decision_function**(*self*, *ypred_proba*, *threshold=0.5*, *X=None*)

Apply a learned threshold to predicted probabilities to get predicted classes.

**Parameters**

- **ypred_proba** (*pd.Series, np.ndarray*) – The classifier's predicted probabilities
- **threshold** (*float, optional*) – Threshold used to make a prediction. Defaults to 0.5.
- **X** (*pd.DataFrame, optional*) – Any extra columns that are needed from training data.

**Returns** predictions

**classmethod is_defined_for_problem_type**(*cls*, *problem_type*)

Returns whether or not an objective is defined for a problem type.

**objective_function**(*self*, *y_true*, *y_predicted*, *X=None*, *sample_weight=None*)

Objective function for Gini coefficient for binary classification.

**optimize_threshold**(*self*, *ypred_proba*, *y_true*, *X=None*)

Learn a binary classification threshold which optimizes the current objective.

**Parameters**

- **ypred_proba** (*pd.Series*) – The classifier's predicted probabilities
- **y_true** (*pd.Series*) – The ground truth for the predictions.
- **X** (*pd.DataFrame, optional*) – Any extra columns that are needed from training data.

**Returns** Optimal threshold for this objective.

**Raises** **RuntimeError** – If objective cannot be optimized.

**positive_only**(*cls*)

If True, this objective is only valid for positive data. Defaults to False.

**score**(*self*, *y_true*, *y_predicted*, *X=None*, *sample_weight=None*)

Returns a numerical score indicating performance based on the differences between the predicted and actual values.

**Parameters**

- **y_predicted** (*pd.Series*) – Predicted values of length [n_samples]
- **y_true** (*pd.Series*) – Actual class labels of length [n_samples]
- **X** (*pd.DataFrame or np.ndarray*) – Extra data of shape [n_samples, n_features] necessary to calculate score
- **sample_weight** (*pd.DataFrame or np.ndarray*) – Sample weights used in computing objective value result

**Returns** score

**validate_inputs**(*self*, *y_true*, *y_predicted*)

Validate inputs for scoring.

**class** evalml.objectives.**LeadScoring**(*true_positives=1*, *false_positives=- 1*)

Lead scoring.

**Parameters**

- **true_positives** (*int*) – Reward for a true positive. Defaults to 1.
- **false_positives** (*int*) – Cost for a false positive. Should be negative. Defaults to -1.

**Attributes**

| | |
|---|---|
| **expected_range** | None |
| **greater_is_better** | True |
| **is_bounded_like_a_percentage** | False |
| **name** | Lead Scoring |
| **perfect_score** | None |
| **problem_types** | [ProblemTypes.BINARY, ProblemTypes.TIME_SERIES_BINARY] |
| **score_needs_proba** | False |

**Methods**

| | |
|---|---|
| *calculate_percent_difference* | Calculate the percent difference between scores. |
| *can_optimize_threshold* | Returns a boolean determining if we can optimize the binary classification objective threshold. |
| *decision_function* | Apply a learned threshold to predicted probabilities to get predicted classes. |
| *is_defined_for_problem_type* | Returns whether or not an objective is defined for a problem type. |
| *objective_function* | Calculate the profit per lead. |
| *optimize_threshold* | Learn a binary classification threshold which optimizes the current objective. |
| *positive_only* | If True, this objective is only valid for positive data. Defaults to False. |
| *score* | Returns a numerical score indicating performance based on the differences between the predicted and actual values. |
| *validate_inputs* | Validate inputs for scoring. |

**classmethod calculate_percent_difference**(*cls*, *score*, *baseline_score*)

Calculate the percent difference between scores.

**Parameters**

- **score** (*float*) – A score. Output of the score method of this objective.
- **baseline_score** (*float*) – A score. Output of the score method of this objective. In practice, this is the score achieved on this objective with a baseline estimator.

**Returns**

**The percent difference between the scores. Note that for objectives that can be interpreted** as percentages, this will be the difference between the reference score and score. For all other objectives, the difference will be normalized by the reference score.

**Return type** float

**property can_optimize_threshold**(*cls*)

Returns a boolean determining if we can optimize the binary classification objective threshold.

This will be false for any objective that works directly with predicted probabilities, like log loss and AUC. Otherwise, it will be true.

**Returns** Whether or not an objective can be optimized.

**Return type** bool

**decision_function**(*self*, *ypred_proba*, *threshold=0.5*, *X=None*)

Apply a learned threshold to predicted probabilities to get predicted classes.

**Parameters**

- **ypred_proba** (*pd.Series, np.ndarray*) – The classifier's predicted probabilities
- **threshold** (*float, optional*) – Threshold used to make a prediction. Defaults to 0.5.
- **X** (*pd.DataFrame, optional*) – Any extra columns that are needed from training data.

**Returns** predictions

**classmethod is_defined_for_problem_type**(*cls*, *problem_type*)

Returns whether or not an objective is defined for a problem type.

**objective_function**(*self*, *y_true*, *y_predicted*, *X=None*, *sample_weight=None*)

Calculate the profit per lead.

**Parameters**

- **y_predicted** (*pd.Series*) – Predicted labels
- **y_true** (*pd.Series*) – True labels
- **X** (*pd.DataFrame*) – Ignored.
- **sample_weight** (*pd.DataFrame*) – Ignored.

**Returns** Profit per lead

**Return type** float

**optimize_threshold**(*self*, *ypred_proba*, *y_true*, *X=None*)

Learn a binary classification threshold which optimizes the current objective.

**Parameters**

- **ypred_proba** (*pd.Series*) – The classifier's predicted probabilities
- **y_true** (*pd.Series*) – The ground truth for the predictions.
- **X** (*pd.DataFrame, optional*) – Any extra columns that are needed from training data.

**Returns** Optimal threshold for this objective.

**Raises** **RuntimeError** – If objective cannot be optimized.

**positive_only**(*cls*)

If True, this objective is only valid for positive data. Defaults to False.

**score**(*self*, *y_true*, *y_predicted*, *X=None*, *sample_weight=None*)

Returns a numerical score indicating performance based on the differences between the predicted and actual values.

**Parameters**

- **y_predicted** (*pd.Series*) – Predicted values of length [n_samples]
- **y_true** (*pd.Series*) – Actual class labels of length [n_samples]
- **X** (*pd.DataFrame or np.ndarray*) – Extra data of shape [n_samples, n_features] necessary to calculate score
- **sample_weight** (*pd.DataFrame or np.ndarray*) – Sample weights used in computing objective value result

**Returns** score

**validate_inputs**(*self*, *y_true*, *y_predicted*)

Validate inputs for scoring.

**class** evalml.objectives.**LogLossBinary**

Log Loss for binary classification.

### Example

```
>>> y_true = pd.Series([0, 1, 1, 0, 0, 1, 1, 1, 0, 1, 1])
>>> y_pred = pd.Series([0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1])
>>> np.testing.assert_almost_equal(LogLossBinary().objective_function(y_true, y_
↪pred), 19.6601745)
```

**Attributes**

| | |
|---|---|
| **expected_range** | [0, 1] |
| **greater_is_better** | False |
| **is_bounded_like** | False |
| **name** | Log Loss Binary |
| **perfect_score** | 0.0 |
| **problem_types** | [ProblemTypes.BINARY, ProblemTypes.TIME_SERIES_BINARY] |
| **score_needs_proba** | True |

**Methods**

| | |
|---|---|
| *calculate_percent_difference* | Calculate the percent difference between scores. |
| *can_optimize_threshold* | Returns a boolean determining if we can optimize the binary classification objective threshold. |
| *decision_function* | Apply a learned threshold to predicted probabilities to get predicted classes. |
| *is_defined_for_problem_type* | Returns whether or not an objective is defined for a problem type. |
| *objective_function* | Objective function for log loss for binary classification. |
| *optimize_threshold* | Learn a binary classification threshold which optimizes the current objective. |
| *positive_only* | If True, this objective is only valid for positive data. Defaults to False. |
| *score* | Returns a numerical score indicating performance based on the differences between the predicted and actual values. |
| *validate_inputs* | Validate inputs for scoring. |

**classmethod calculate_percent_difference**(*cls*, *score*, *baseline_score*)

Calculate the percent difference between scores.

**Parameters**

- **score** (*float*) – A score. Output of the score method of this objective.
- **baseline_score** (*float*) – A score. Output of the score method of this objective. In practice, this is the score achieved on this objective with a baseline estimator.

**Returns**

**The percent difference between the scores. Note that for objectives that can be interpreted** as percentages, this will be the difference between the reference score and score. For all other objectives, the difference will be normalized by the reference score.

**Return type** float

**property can_optimize_threshold**(*cls*)

Returns a boolean determining if we can optimize the binary classification objective threshold.

This will be false for any objective that works directly with predicted probabilities, like log loss and AUC. Otherwise, it will be true.

**Returns** Whether or not an objective can be optimized.

**Return type** bool

**decision_function**(*self*, *ypred_proba*, *threshold=0.5*, *X=None*)

Apply a learned threshold to predicted probabilities to get predicted classes.

**Parameters**

- **ypred_proba** (*pd.Series, np.ndarray*) – The classifier's predicted probabilities
- **threshold** (*float, optional*) – Threshold used to make a prediction. Defaults to 0.5.
- **X** (*pd.DataFrame, optional*) – Any extra columns that are needed from training data.

**Returns** predictions

**classmethod is_defined_for_problem_type**(*cls*, *problem_type*)

Returns whether or not an objective is defined for a problem type.

**objective_function**(*self*, *y_true*, *y_predicted*, *X=None*, *sample_weight=None*)

Objective function for log loss for binary classification.

**optimize_threshold**(*self*, *ypred_proba*, *y_true*, *X=None*)

Learn a binary classification threshold which optimizes the current objective.

**Parameters**

- **ypred_proba** (*pd.Series*) – The classifier's predicted probabilities
- **y_true** (*pd.Series*) – The ground truth for the predictions.
- **X** (*pd.DataFrame, optional*) – Any extra columns that are needed from training data.

**Returns** Optimal threshold for this objective.

**Raises** **RuntimeError** – If objective cannot be optimized.

**positive_only**(*cls*)

If True, this objective is only valid for positive data. Defaults to False.

**score**(*self*, *y_true*, *y_predicted*, *X=None*, *sample_weight=None*)

Returns a numerical score indicating performance based on the differences between the predicted and actual values.

**Parameters**

- **y_predicted** (*pd.Series*) – Predicted values of length [n_samples]
- **y_true** (*pd.Series*) – Actual class labels of length [n_samples]
- **X** (*pd.DataFrame or np.ndarray*) – Extra data of shape [n_samples, n_features] necessary to calculate score
- **sample_weight** (*pd.DataFrame or np.ndarray*) – Sample weights used in computing objective value result

**Returns** score

**validate_inputs**(*self*, *y_true*, *y_predicted*)

Validate inputs for scoring.

**class** evalml.objectives.**LogLossMulticlass**

Log Loss for multiclass classification.

### Example

```
>>> y_true = [0, 1, 2, 0, 2, 1]
>>> y_pred = [[0.7, 0.2, 0.1],
...           [0.3, 0.5, 0.2],
...           [0.1, 0.3, 0.6],
...           [0.9, 0.1, 0.0],
...           [0.3, 0.1, 0.6],
...           [0.5, 0.5, 0.0]]
>>> np.testing.assert_almost_equal(LogLossMulticlass().objective_function(y_true, y_
↪pred), 0.4783301)
```

**Attributes**

| | |
|---|---|
| **expected_range** | [0, 1] |
| **greater_is_better** | False |
| **is_bounded_like_a_percentage** | False |
| **name** | Log Loss Multiclass |
| **perfect_score** | 0.0 |
| **problem_types** | [ProblemTypes.MULTICLASS, ProblemTypes.TIME_SERIES_MULTICLASS] |
| **score_needs_proba** | True |

**Methods**

| | |
|---|---|
| *calculate_percent_difference* | Calculate the percent difference between scores. |
| *is_defined_for_problem_type* | Returns whether or not an objective is defined for a problem type. |
| *objective_function* | Objective function for log loss for multiclass classification. |
| *positive_only* | If True, this objective is only valid for positive data. Defaults to False. |
| *score* | Returns a numerical score indicating performance based on the differences between the predicted and actual values. |
| *validate_inputs* | Validates the input based on a few simple checks. |

**classmethod calculate_percent_difference**(*cls*, *score*, *baseline_score*)

Calculate the percent difference between scores.

**Parameters**

- **score** (*float*) – A score. Output of the score method of this objective.
- **baseline_score** (*float*) – A score. Output of the score method of this objective. In practice, this is the score achieved on this objective with a baseline estimator.

**Returns**

**The percent difference between the scores. Note that for objectives that can be interpreted** as percentages, this will be the difference between the reference score and score. For all other objectives, the difference will be normalized by the reference score.

**Return type** float

**classmethod is_defined_for_problem_type**(*cls*, *problem_type*)

Returns whether or not an objective is defined for a problem type.

**objective_function**(*self*, *y_true*, *y_predicted*, *X=None*, *sample_weight=None*)

Objective function for log loss for multiclass classification.

**positive_only**(*cls*)

If True, this objective is only valid for positive data. Defaults to False.

**score**(*self*, *y_true*, *y_predicted*, *X=None*, *sample_weight=None*)

Returns a numerical score indicating performance based on the differences between the predicted and actual values.

**Parameters**

- **y_predicted** (*pd.Series*) – Predicted values of length [n_samples]
- **y_true** (*pd.Series*) – Actual class labels of length [n_samples]
- **X** (*pd.DataFrame or np.ndarray*) – Extra data of shape [n_samples, n_features] necessary to calculate score
- **sample_weight** (*pd.DataFrame or np.ndarray*) – Sample weights used in computing objective value result

**Returns** score

**validate_inputs**(*self*, *y_true*, *y_predicted*)

Validates the input based on a few simple checks.

**Parameters**

- **y_predicted** (*pd.Series, or pd.DataFrame*) – Predicted values of length [n_samples].
- **y_true** (*pd.Series*) – Actual class labels of length [n_samples].

**Raises** **ValueError** – If the inputs are malformed.

**class** evalml.objectives.**MAE**

Mean absolute error for regression.

### Example

```
>>> y_true = pd.Series([1.5, 2, 3, 1, 0.5, 1, 2.5, 2.5, 1, 0.5, 2])
>>> y_pred = pd.Series([1.5, 2.5, 2, 1, 0.5, 1, 3, 2.25, 0.75, 0.25, 1.75])
>>> np.testing.assert_almost_equal(MAE().objective_function(y_true, y_pred), 0.
↪2727272)
```

**Attributes**

| | |
|---|---|
| **expected_range** | None |
| **greater_is_better** | False |
| **is_bounded_like_percentage** | True |
| **name** | MAE |
| **perfect_score** | 0.0 |
| **problem_types** | [ProblemTypes.REGRESSION, ProblemTypes.TIME_SERIES_REGRESSION] |
| **score_needs_proba** | False |

**Methods**

| | |
|---|---|
| *calculate_percent_difference* | Calculate the percent difference between scores. |
| *is_defined_for_problem_type* | Returns whether or not an objective is defined for a problem type. |
| *objective_function* | Objective function for mean absolute error for regression. |
| *positive_only* | If True, this objective is only valid for positive data. Defaults to False. |
| *score* | Returns a numerical score indicating performance based on the differences between the predicted and actual values. |
| *validate_inputs* | Validates the input based on a few simple checks. |

**classmethod calculate_percent_difference**(*cls*, *score*, *baseline_score*)

Calculate the percent difference between scores.

**Parameters**

- **score** (*float*) – A score. Output of the score method of this objective.
- **baseline_score** (*float*) – A score. Output of the score method of this objective. In practice, this is the score achieved on this objective with a baseline estimator.

**Returns**

**The percent difference between the scores. Note that for objectives that can be interpreted** as percentages, this will be the difference between the reference score and score. For all other objectives, the difference will be normalized by the reference score.

**Return type** float

**classmethod is_defined_for_problem_type**(*cls*, *problem_type*)

Returns whether or not an objective is defined for a problem type.

**objective_function**(*self*, *y_true*, *y_predicted*, *X=None*, *sample_weight=None*)

Objective function for mean absolute error for regression.

**positive_only**(*cls*)

If True, this objective is only valid for positive data. Defaults to False.

**score**(*self*, *y_true*, *y_predicted*, *X=None*, *sample_weight=None*)

Returns a numerical score indicating performance based on the differences between the predicted and actual values.

**Parameters**

- **y_predicted** (*pd.Series*) – Predicted values of length [n_samples]
- **y_true** (*pd.Series*) – Actual class labels of length [n_samples]
- **X** (*pd.DataFrame or np.ndarray*) – Extra data of shape [n_samples, n_features] necessary to calculate score
- **sample_weight** (*pd.DataFrame or np.ndarray*) – Sample weights used in computing objective value result

**Returns** score

**validate_inputs**(*self*, *y_true*, *y_predicted*)

Validates the input based on a few simple checks.

**Parameters**

- **y_predicted** (*pd.Series, or pd.DataFrame*) – Predicted values of length [n_samples].
- **y_true** (*pd.Series*) – Actual class labels of length [n_samples].

**Raises** **ValueError** – If the inputs are malformed.

**class** evalml.objectives.**MAPE**

Mean absolute percentage error for time series regression. Scaled by 100 to return a percentage.

Only valid for nonzero inputs. Otherwise, will throw a ValueError.

### Example

```
>>> y_true = pd.Series([1.5, 2, 3, 1, 0.5, 1, 2.5, 2.5, 1, 0.5, 2])
>>> y_pred = pd.Series([1.5, 2.5, 2, 1, 0.5, 1, 3, 2.25, 0.75, 0.25, 1.75])
>>> np.testing.assert_almost_equal(MAPE().objective_function(y_true, y_pred), 15.
↪9848484)
```

**Attributes**

| | |
|---|---|
| **expected_range** | None |
| **greater_is_better** | False |
| **is_bounded_like_percentage** | False |
| **name** | Mean Absolute Percentage Error |
| **perfect_score** | 0.0 |
| **problem_types** | [ProblemTypes.TIME_SERIES_REGRESSION] |
| **score_needs_proba** | False |

**Methods**

| | |
|---|---|
| *calculate_percent_difference* | Calculate the percent difference between scores. |
| *is_defined_for_problem_type* | Returns whether or not an objective is defined for a problem type. |
| *objective_function* | Objective function for mean absolute percentage error for time series regression. |
| *positive_only* | If True, this objective is only valid for positive data. |
| *score* | Returns a numerical score indicating performance based on the differences between the predicted and actual values. |
| *validate_inputs* | Validates the input based on a few simple checks. |

**classmethod calculate_percent_difference**(*cls*, *score*, *baseline_score*)

Calculate the percent difference between scores.

**Parameters**

- **score** (*float*) – A score. Output of the score method of this objective.

- **baseline_score** (*float*) – A score. Output of the score method of this objective. In practice, this is the score achieved on this objective with a baseline estimator.

**Returns**

**The percent difference between the scores. Note that for objectives that can be interpreted** as percentages, this will be the difference between the reference score and score. For all other objectives, the difference will be normalized by the reference score.

**Return type** float

**classmethod is_defined_for_problem_type**(*cls*, *problem_type*)

Returns whether or not an objective is defined for a problem type.

**objective_function**(*self*, *y_true*, *y_predicted*, *X=None*, *sample_weight=None*)

Objective function for mean absolute percentage error for time series regression.

**positive_only**(*self*)

If True, this objective is only valid for positive data.

**score**(*self*, *y_true*, *y_predicted*, *X=None*, *sample_weight=None*)

Returns a numerical score indicating performance based on the differences between the predicted and actual values.

**Parameters**

- **y_predicted** (*pd.Series*) – Predicted values of length [n_samples]
- **y_true** (*pd.Series*) – Actual class labels of length [n_samples]
- **X** (*pd.DataFrame or np.ndarray*) – Extra data of shape [n_samples, n_features] necessary to calculate score
- **sample_weight** (*pd.DataFrame or np.ndarray*) – Sample weights used in computing objective value result

**Returns** score

**validate_inputs**(*self*, *y_true*, *y_predicted*)

Validates the input based on a few simple checks.

**Parameters**

- **y_predicted** (*pd.Series, or pd.DataFrame*) – Predicted values of length [n_samples].
- **y_true** (*pd.Series*) – Actual class labels of length [n_samples].

**Raises** **ValueError** – If the inputs are malformed.

**class** evalml.objectives.**MaxError**

Maximum residual error for regression.

### Example

```
>>> y_true = pd.Series([1.5, 2, 3, 1, 0.5, 1, 2.5, 2.5, 1, 0.5, 2])
>>> y_pred = pd.Series([1.5, 2.5, 2, 1, 0.5, 1, 3, 2.25, 0.75, 0.25, 1.75])
>>> np.testing.assert_almost_equal(MaxError().objective_function(y_true, y_pred), 1.
↪0)
```

**Attributes**

| | |
|---|---|
| **expected_range** | None |
| **greater_is_better** | False |
| **is_bounded_like_percentage** | False |
| **name** | MaxError |
| **perfect_score** | 0.0 |
| **problem_types** | [ProblemTypes.REGRESSION, ProblemTypes.TIME_SERIES_REGRESSION] |
| **score_needs_proba** | False |

**Methods**

| | |
|---|---|
| *calculate_percent_difference* | Calculate the percent difference between scores. |
| *is_defined_for_problem_type* | Returns whether or not an objective is defined for a problem type. |
| *objective_function* | Objective function for maximum residual error for regression. |
| *positive_only* | If True, this objective is only valid for positive data. Defaults to False. |
| *score* | Returns a numerical score indicating performance based on the differences between the predicted and actual values. |
| *validate_inputs* | Validates the input based on a few simple checks. |

**classmethod calculate_percent_difference**(*cls*, *score*, *baseline_score*)

Calculate the percent difference between scores.

**Parameters**

- **score** (*float*) – A score. Output of the score method of this objective.
- **baseline_score** (*float*) – A score. Output of the score method of this objective. In practice, this is the score achieved on this objective with a baseline estimator.

**Returns**

**The percent difference between the scores. Note that for objectives that can be interpreted** as percentages, this will be the difference between the reference score and score. For all other objectives, the difference will be normalized by the reference score.

**Return type** float

**classmethod is_defined_for_problem_type**(*cls*, *problem_type*)

Returns whether or not an objective is defined for a problem type.

**objective_function**(*self*, *y_true*, *y_predicted*, *X=None*, *sample_weight=None*)

Objective function for maximum residual error for regression.

**positive_only**(*cls*)

If True, this objective is only valid for positive data. Defaults to False.

**score**(*self*, *y_true*, *y_predicted*, *X=None*, *sample_weight=None*)

Returns a numerical score indicating performance based on the differences between the predicted and actual values.

**Parameters**

- **y_predicted** (*pd.Series*) – Predicted values of length [n_samples]
- **y_true** (*pd.Series*) – Actual class labels of length [n_samples]
- **X** (*pd.DataFrame or np.ndarray*) – Extra data of shape [n_samples, n_features] necessary to calculate score
- **sample_weight** (*pd.DataFrame or np.ndarray*) – Sample weights used in computing objective value result

**Returns** score

**validate_inputs**(*self*, *y_true*, *y_predicted*)

Validates the input based on a few simple checks.

**Parameters**

- **y_predicted** (*pd.Series, or pd.DataFrame*) – Predicted values of length [n_samples].
- **y_true** (*pd.Series*) – Actual class labels of length [n_samples].

**Raises** **ValueError** – If the inputs are malformed.

**class** evalml.objectives.**MCCBinary**

Matthews correlation coefficient for binary classification.

### Example

```
>>> y_true = pd.Series([0, 1, 1, 0, 0, 1, 1, 1, 0, 1, 1])
>>> y_pred = pd.Series([0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1])
>>> np.testing.assert_almost_equal(MCCBinary().objective_function(y_true, y_pred),␣
↪0.2390457)
```

**Attributes**

| | |
|---|---|
| **expected_range** | None |
| **greater_is_better** | True |
| **is_bounded_like_a_percentage** | False |
| **name** | MCC Binary |
| **perfect_score** | 1.0 |
| **problem_types** | [ProblemTypes.BINARY, ProblemTypes.TIME_SERIES_BINARY] |
| **score_needs_proba** | False |

**Methods**

| | |
|---|---|
| *calculate_percent_difference* | Calculate the percent difference between scores. |
| *can_optimize_threshold* | Returns a boolean determining if we can optimize the binary classification objective threshold. |
| *decision_function* | Apply a learned threshold to predicted probabilities to get predicted classes. |
| *is_defined_for_problem_type* | Returns whether or not an objective is defined for a problem type. |
| *objective_function* | Objective function for Matthews correlation coefficient for binary classification. |
| *optimize_threshold* | Learn a binary classification threshold which optimizes the current objective. |
| *positive_only* | If True, this objective is only valid for positive data. Defaults to False. |
| *score* | Returns a numerical score indicating performance based on the differences between the predicted and actual values. |
| *validate_inputs* | Validate inputs for scoring. |

**classmethod calculate_percent_difference**(*cls*, *score*, *baseline_score*)

Calculate the percent difference between scores.

**Parameters**

- **score** (*float*) – A score. Output of the score method of this objective.
- **baseline_score** (*float*) – A score. Output of the score method of this objective. In practice, this is the score achieved on this objective with a baseline estimator.

**Returns**

**The percent difference between the scores. Note that for objectives that can be interpreted** as percentages, this will be the difference between the reference score and score. For all other objectives, the difference will be normalized by the reference score.

**Return type** float

**property can_optimize_threshold**(*cls*)

Returns a boolean determining if we can optimize the binary classification objective threshold.

This will be false for any objective that works directly with predicted probabilities, like log loss and AUC. Otherwise, it will be true.

**Returns** Whether or not an objective can be optimized.

**Return type** bool

**decision_function**(*self*, *ypred_proba*, *threshold=0.5*, *X=None*)

Apply a learned threshold to predicted probabilities to get predicted classes.

**Parameters**

- **ypred_proba** (*pd.Series, np.ndarray*) – The classifier's predicted probabilities
- **threshold** (*float, optional*) – Threshold used to make a prediction. Defaults to 0.5.
- **X** (*pd.DataFrame, optional*) – Any extra columns that are needed from training data.

**Returns** predictions

**classmethod is_defined_for_problem_type**(*cls*, *problem_type*)

Returns whether or not an objective is defined for a problem type.

**objective_function**(*self*, *y_true*, *y_predicted*, *X=None*, *sample_weight=None*)

Objective function for Matthews correlation coefficient for binary classification.

**optimize_threshold**(*self*, *ypred_proba*, *y_true*, *X=None*)

Learn a binary classification threshold which optimizes the current objective.

**Parameters**

- **ypred_proba** (*pd.Series*) – The classifier's predicted probabilities
- **y_true** (*pd.Series*) – The ground truth for the predictions.
- **X** (*pd.DataFrame, optional*) – Any extra columns that are needed from training data.

**Returns** Optimal threshold for this objective.

**Raises** **RuntimeError** – If objective cannot be optimized.

**positive_only**(*cls*)

If True, this objective is only valid for positive data. Defaults to False.

**score**(*self*, *y_true*, *y_predicted*, *X=None*, *sample_weight=None*)

Returns a numerical score indicating performance based on the differences between the predicted and actual values.

**Parameters**

- **y_predicted** (*pd.Series*) – Predicted values of length [n_samples]
- **y_true** (*pd.Series*) – Actual class labels of length [n_samples]
- **X** (*pd.DataFrame or np.ndarray*) – Extra data of shape [n_samples, n_features] necessary to calculate score
- **sample_weight** (*pd.DataFrame or np.ndarray*) – Sample weights used in computing objective value result

**Returns** score

**validate_inputs**(*self*, *y_true*, *y_predicted*)

Validate inputs for scoring.

**class** evalml.objectives.**MCCMulticlass**

Matthews correlation coefficient for multiclass classification.

### Example

```
>>> y_true = pd.Series([0, 1, 0, 2, 0, 1, 2, 1, 2, 0, 2])
>>> y_pred = pd.Series([0, 0, 0, 0, 1, 1, 1, 1, 2, 2, 2])
>>> np.testing.assert_almost_equal(MCCMulticlass().objective_function(y_true, y_
↪pred), 0.325)
```

**Attributes**

| | |
|---|---|
| **expected_range** | None |
| **greater_is_better** | True |
| **is_bounded_like** | False |
| **name** | MCC Multiclass |
| **perfect_score** | 1.0 |
| **problem_types** | [ProblemTypes.MULTICLASS, ProblemTypes.TIME_SERIES_MULTICLASS] |
| **score_needs_proba** | False |

**Methods**

| | |
|---|---|
| *calculate_percent_difference* | Calculate the percent difference between scores. |
| *is_defined_for_problem_type* | Returns whether or not an objective is defined for a problem type. |
| *objective_function* | Objective function for Matthews correlation coefficient for multiclass classification. |
| *positive_only* | If True, this objective is only valid for positive data. Defaults to False. |
| *score* | Returns a numerical score indicating performance based on the differences between the predicted and actual values. |
| *validate_inputs* | Validates the input based on a few simple checks. |

**classmethod calculate_percent_difference**(*cls*, *score*, *baseline_score*)

Calculate the percent difference between scores.

**Parameters**

- **score** (*float*) – A score. Output of the score method of this objective.
- **baseline_score** (*float*) – A score. Output of the score method of this objective. In practice, this is the score achieved on this objective with a baseline estimator.

**Returns**

**The percent difference between the scores. Note that for objectives that can be interpreted** as percentages, this will be the difference between the reference score and score. For all other objectives, the difference will be normalized by the reference score.

**Return type** float

**classmethod is_defined_for_problem_type**(*cls*, *problem_type*)

Returns whether or not an objective is defined for a problem type.

**objective_function**(*self*, *y_true*, *y_predicted*, *X=None*, *sample_weight=None*)

Objective function for Matthews correlation coefficient for multiclass classification.

**positive_only**(*cls*)

If True, this objective is only valid for positive data. Defaults to False.

**score**(*self*, *y_true*, *y_predicted*, *X=None*, *sample_weight=None*)

Returns a numerical score indicating performance based on the differences between the predicted and actual values.

**Parameters**

- **y_predicted** (*pd.Series*) – Predicted values of length [n_samples]
- **y_true** (*pd.Series*) – Actual class labels of length [n_samples]
- **X** (*pd.DataFrame or np.ndarray*) – Extra data of shape [n_samples, n_features] necessary to calculate score
- **sample_weight** (*pd.DataFrame or np.ndarray*) – Sample weights used in computing objective value result

**Returns** score

**validate_inputs**(*self*, *y_true*, *y_predicted*)

Validates the input based on a few simple checks.

**Parameters**

- **y_predicted** (*pd.Series, or pd.DataFrame*) – Predicted values of length [n_samples].
- **y_true** (*pd.Series*) – Actual class labels of length [n_samples].

**Raises** **ValueError** – If the inputs are malformed.

**class** evalml.objectives.**MeanSquaredLogError**

Mean squared log error for regression.

Only valid for nonnegative inputs. Otherwise, will throw a ValueError.

### Example

```
>>> y_true = pd.Series([1.5, 2, 3, 1, 0.5, 1, 2.5, 2.5, 1, 0.5, 2])
>>> y_pred = pd.Series([1.5, 2.5, 2, 1, 0.5, 1, 3, 2.25, 0.75, 0.25, 1.75])
>>> np.testing.assert_almost_equal(MeanSquaredLogError().objective_function(y_true,␣
↪y_pred), 0.0171353)
```

**Attributes**

| | |
|---|---|
| **expected_range** | None |
| **greater_is_better** | False |
| **is_bounded_like_percentage** | False |
| **name** | Mean Squared Log Error |
| **perfect_score** | 0.0 |
| **problem_types** | [ProblemTypes.REGRESSION, ProblemTypes.TIME_SERIES_REGRESSION] |
| **score_needs_proba** | False |

**Methods**

| | |
|---|---|
| *calculate_percent_difference* | Calculate the percent difference between scores. |
| *is_defined_for_problem_type* | Returns whether or not an objective is defined for a problem type. |
| *objective_function* | Objective function for mean squared log error for regression. |
| *positive_only* | If True, this objective is only valid for positive data. |
| *score* | Returns a numerical score indicating performance based on the differences between the predicted and actual values. |
| *validate_inputs* | Validates the input based on a few simple checks. |

**classmethod calculate_percent_difference**(*cls*, *score*, *baseline_score*)

Calculate the percent difference between scores.

**Parameters**

- **score** (*float*) – A score. Output of the score method of this objective.
- **baseline_score** (*float*) – A score. Output of the score method of this objective. In practice, this is the score achieved on this objective with a baseline estimator.

**Returns**

**The percent difference between the scores. Note that for objectives that can be interpreted** as percentages, this will be the difference between the reference score and score. For all other objectives, the difference will be normalized by the reference score.

**Return type** float

**classmethod is_defined_for_problem_type**(*cls*, *problem_type*)

Returns whether or not an objective is defined for a problem type.

**objective_function**(*self*, *y_true*, *y_predicted*, *X=None*, *sample_weight=None*)

Objective function for mean squared log error for regression.

**positive_only**(*self*)

If True, this objective is only valid for positive data.

**score**(*self*, *y_true*, *y_predicted*, *X=None*, *sample_weight=None*)

Returns a numerical score indicating performance based on the differences between the predicted and actual values.

**Parameters**

- **y_predicted** (*pd.Series*) – Predicted values of length [n_samples]
- **y_true** (*pd.Series*) – Actual class labels of length [n_samples]
- **X** (*pd.DataFrame or np.ndarray*) – Extra data of shape [n_samples, n_features] necessary to calculate score
- **sample_weight** (*pd.DataFrame or np.ndarray*) – Sample weights used in computing objective value result

**Returns** score

**validate_inputs**(*self*, *y_true*, *y_predicted*)

Validates the input based on a few simple checks.

**Parameters**

- **y_predicted** (*pd.Series, or pd.DataFrame*) – Predicted values of length [n_samples].
- **y_true** (*pd.Series*) – Actual class labels of length [n_samples].

**Raises** **ValueError** – If the inputs are malformed.

**class** evalml.objectives.**MedianAE**

Median absolute error for regression.

### Example

```
>>> y_true = pd.Series([1.5, 2, 3, 1, 0.5, 1, 2.5, 2.5, 1, 0.5, 2])
>>> y_pred = pd.Series([1.5, 2.5, 2, 1, 0.5, 1, 3, 2.25, 0.75, 0.25, 1.75])
>>> np.testing.assert_almost_equal(MedianAE().objective_function(y_true, y_pred), 0.
↪25)
```

**Attributes**

| | |
|---|---|
| **expected_range** | None |
| **greater_is_better** | False |
| **is_bounded_like_a_percentage** | False |
| **name** | MedianAE |
| **perfect_score** | 0.0 |
| **problem_types** | [ProblemTypes.REGRESSION, ProblemTypes.TIME_SERIES_REGRESSION] |
| **score_needs_proba** | False |

**Methods**

| | |
|---|---|
| *calculate_percent_difference* | Calculate the percent difference between scores. |
| *is_defined_for_problem_type* | Returns whether or not an objective is defined for a problem type. |
| *objective_function* | Objective function for median absolute error for regression. |
| *positive_only* | If True, this objective is only valid for positive data. Defaults to False. |
| *score* | Returns a numerical score indicating performance based on the differences between the predicted and actual values. |
| *validate_inputs* | Validates the input based on a few simple checks. |

**classmethod calculate_percent_difference**(*cls*, *score*, *baseline_score*)

Calculate the percent difference between scores.

**Parameters**

- **score** (*float*) – A score. Output of the score method of this objective.
- **baseline_score** (*float*) – A score. Output of the score method of this objective. In practice, this is the score achieved on this objective with a baseline estimator.

**Returns**

**The percent difference between the scores. Note that for objectives that can be interpreted** as percentages, this will be the difference between the reference score and score. For all other objectives, the difference will be normalized by the reference score.

**Return type** float

**classmethod is_defined_for_problem_type**(*cls*, *problem_type*)

Returns whether or not an objective is defined for a problem type.

**objective_function**(*self*, *y_true*, *y_predicted*, *X=None*, *sample_weight=None*)

Objective function for median absolute error for regression.

**positive_only**(*cls*)

If True, this objective is only valid for positive data. Defaults to False.

**score**(*self*, *y_true*, *y_predicted*, *X=None*, *sample_weight=None*)

Returns a numerical score indicating performance based on the differences between the predicted and actual values.

**Parameters**

- **y_predicted** (*pd.Series*) – Predicted values of length [n_samples]
- **y_true** (*pd.Series*) – Actual class labels of length [n_samples]
- **X** (*pd.DataFrame or np.ndarray*) – Extra data of shape [n_samples, n_features] necessary to calculate score
- **sample_weight** (*pd.DataFrame or np.ndarray*) – Sample weights used in computing objective value result

**Returns** score

**validate_inputs**(*self*, *y_true*, *y_predicted*)

Validates the input based on a few simple checks.

**Parameters**

- **y_predicted** (*pd.Series, or pd.DataFrame*) – Predicted values of length [n_samples].
- **y_true** (*pd.Series*) – Actual class labels of length [n_samples].

**Raises** **ValueError** – If the inputs are malformed.

**class** evalml.objectives.**MSE**

Mean squared error for regression.

### Example

```
>>> y_true = pd.Series([1.5, 2, 3, 1, 0.5, 1, 2.5, 2.5, 1, 0.5, 2])
>>> y_pred = pd.Series([1.5, 2.5, 2, 1, 0.5, 1, 3, 2.25, 0.75, 0.25, 1.75])
>>> np.testing.assert_almost_equal(MSE().objective_function(y_true, y_pred), 0.
↪1590909)
```

**Attributes**

| | |
|---|---|
| **expected_range** | None |
| **greater_is_better** | False |
| **is_bounded_like_percentage** | False |
| **name** | MSE |
| **perfect_score** | 0.0 |
| **problem_types** | [ProblemTypes.REGRESSION, ProblemTypes.TIME_SERIES_REGRESSION] |
| **score_needs_proba** | False |

**Methods**

| | |
|---|---|
| *calculate_percent_difference* | Calculate the percent difference between scores. |
| *is_defined_for_problem_type* | Returns whether or not an objective is defined for a problem type. |
| *objective_function* | Objective function for mean squared error for regression. |
| *positive_only* | If True, this objective is only valid for positive data. Defaults to False. |
| *score* | Returns a numerical score indicating performance based on the differences between the predicted and actual values. |
| *validate_inputs* | Validates the input based on a few simple checks. |

**classmethod calculate_percent_difference**(*cls*, *score*, *baseline_score*)

Calculate the percent difference between scores.

**Parameters**

- **score** (*float*) – A score. Output of the score method of this objective.
- **baseline_score** (*float*) – A score. Output of the score method of this objective. In practice, this is the score achieved on this objective with a baseline estimator.

**Returns**

**The percent difference between the scores. Note that for objectives that can be interpreted** as percentages, this will be the difference between the reference score and score. For all other objectives, the difference will be normalized by the reference score.

**Return type** float

**classmethod is_defined_for_problem_type**(*cls*, *problem_type*)

Returns whether or not an objective is defined for a problem type.

**objective_function**(*self*, *y_true*, *y_predicted*, *X=None*, *sample_weight=None*)

Objective function for mean squared error for regression.

**positive_only**(*cls*)

If True, this objective is only valid for positive data. Defaults to False.

**score**(*self*, *y_true*, *y_predicted*, *X=None*, *sample_weight=None*)

Returns a numerical score indicating performance based on the differences between the predicted and actual values.

**Parameters**

- **y_predicted** (*pd.Series*) – Predicted values of length [n_samples]
- **y_true** (*pd.Series*) – Actual class labels of length [n_samples]
- **X** (*pd.DataFrame or np.ndarray*) – Extra data of shape [n_samples, n_features] necessary to calculate score
- **sample_weight** (*pd.DataFrame or np.ndarray*) – Sample weights used in computing objective value result

**Returns** score

**validate_inputs**(*self*, *y_true*, *y_predicted*)

Validates the input based on a few simple checks.

**Parameters**

- **y_predicted** (*pd.Series, or pd.DataFrame*) – Predicted values of length [n_samples].
- **y_true** (*pd.Series*) – Actual class labels of length [n_samples].

**Raises** **ValueError** – If the inputs are malformed.

**class** evalml.objectives.**MulticlassClassificationObjective**

Base class for all multiclass classification objectives.

**Attributes**

| | |
|---|---|
| **problem_types** | [ProblemTypes.MULTICLASS, ProblemTypes.TIME_SERIES_MULTICLASS] |

**Methods**

| | |
|---|---|
| *calculate_percent_difference* | Calculate the percent difference between scores. |
| *expected_range* | Returns the expected range of the objective, which is not necessarily the possible ranges. |
| *greater_is_better* | Returns a boolean determining if a greater score indicates better model performance. |
| *is_bounded_like_percentage* | Returns whether this objective is bounded between 0 and 1, inclusive. |
| *is_defined_for_problem_type* | Returns whether or not an objective is defined for a problem type. |
| *name* | Returns a name describing the objective. |
| *objective_function* | Computes the relative value of the provided predictions compared to the actual labels, according a specified metric. |
| *perfect_score* | Returns the score obtained by evaluating this objective on a perfect model. |
| *positive_only* | If True, this objective is only valid for positive data. Defaults to False. |
| *score* | Returns a numerical score indicating performance based on the differences between the predicted and actual values. |
| *score_needs_proba* | Returns a boolean determining if the score() method needs probability estimates. |
| *validate_inputs* | Validates the input based on a few simple checks. |

**classmethod calculate_percent_difference**(*cls*, *score*, *baseline_score*)

Calculate the percent difference between scores.

**Parameters**

- **score** (*float*) – A score. Output of the score method of this objective.
- **baseline_score** (*float*) – A score. Output of the score method of this objective. In practice, this is the score achieved on this objective with a baseline estimator.

**Returns**

**The percent difference between the scores. Note that for objectives that can be interpreted** as percentages, this will be the difference between the reference score and score. For all other objectives, the difference will be normalized by the reference score.

**Return type** float

**property expected_range**(*cls*)

Returns the expected range of the objective, which is not necessarily the possible ranges.

For example, our expected R2 range is from [-1, 1], although the actual range is (-inf, 1].

**property greater_is_better**(*cls*)

Returns a boolean determining if a greater score indicates better model performance.

**property is_bounded_like_percentage**(*cls*)

Returns whether this objective is bounded between 0 and 1, inclusive.

**classmethod is_defined_for_problem_type**(*cls*, *problem_type*)

Returns whether or not an objective is defined for a problem type.

**property name**(*cls*)

Returns a name describing the objective.

**abstract classmethod objective_function**(*cls*, *y_true*, *y_predicted*, *X=None*, *sample_weight=None*)

Computes the relative value of the provided predictions compared to the actual labels, according a specified metric.

**Parameters**

- **y_predicted** (*pd.Series*) – Predicted values of length [n_samples]
- **y_true** (*pd.Series*) – Actual class labels of length [n_samples]
- **X** (*pd.DataFrame or np.ndarray*) – Extra data of shape [n_samples, n_features] necessary to calculate score
- **sample_weight** (*pd.DataFrame or np.ndarray*) – Sample weights used in computing objective value result

**Returns** Numerical value used to calculate score

**property perfect_score**(*cls*)

Returns the score obtained by evaluating this objective on a perfect model.

**positive_only**(*cls*)

If True, this objective is only valid for positive data. Defaults to False.

**score**(*self*, *y_true*, *y_predicted*, *X=None*, *sample_weight=None*)

Returns a numerical score indicating performance based on the differences between the predicted and actual values.

**Parameters**

- **y_predicted** (*pd.Series*) – Predicted values of length [n_samples]
- **y_true** (*pd.Series*) – Actual class labels of length [n_samples]
- **X** (*pd.DataFrame or np.ndarray*) – Extra data of shape [n_samples, n_features] necessary to calculate score
- **sample_weight** (*pd.DataFrame or np.ndarray*) – Sample weights used in computing objective value result

**Returns** score

**property score_needs_proba**(*cls*)

Returns a boolean determining if the score() method needs probability estimates.

This should be true for objectives which work with predicted probabilities, like log loss or AUC, and false for objectives which compare predicted class labels to the actual labels, like F1 or correlation.

**validate_inputs**(*self*, *y_true*, *y_predicted*)

Validates the input based on a few simple checks.

**Parameters**

- **y_predicted** (*pd.Series, or pd.DataFrame*) – Predicted values of length [n_samples].
- **y_true** (*pd.Series*) – Actual class labels of length [n_samples].

**Raises** **ValueError** – If the inputs are malformed.

**class** evalml.objectives.**ObjectiveBase**

Base class for all objectives.

**Attributes**

| | |
|---|---|
| **problem_types** | None |

**Methods**

| | |
|---|---|
| *calculate_percent_difference* | Calculate the percent difference between scores. |
| *expected_range* | Returns the expected range of the objective, which is not necessarily the possible ranges. |
| *greater_is_better* | Returns a boolean determining if a greater score indicates better model performance. |
| *is_bounded_like_percentage* | Returns whether this objective is bounded between 0 and 1, inclusive. |
| *is_defined_for_problem_type* | Returns whether or not an objective is defined for a problem type. |
| *name* | Returns a name describing the objective. |
| *objective_function* | Computes the relative value of the provided predictions compared to the actual labels, according a specified metric. |
| *perfect_score* | Returns the score obtained by evaluating this objective on a perfect model. |
| *positive_only* | If True, this objective is only valid for positive data. Defaults to False. |
| *score* | Returns a numerical score indicating performance based on the differences between the predicted and actual values. |
| *score_needs_proba* | Returns a boolean determining if the score() method needs probability estimates. |
| *validate_inputs* | Validates the input based on a few simple checks. |

**classmethod calculate_percent_difference**(*cls*, *score*, *baseline_score*)

Calculate the percent difference between scores.

**Parameters**

- **score** (*float*) – A score. Output of the score method of this objective.
- **baseline_score** (*float*) – A score. Output of the score method of this objective. In practice, this is the score achieved on this objective with a baseline estimator.

**Returns**

**The percent difference between the scores. Note that for objectives that can be interpreted** as percentages, this will be the difference between the reference score and score. For all other objectives, the difference will be normalized by the reference score.

**Return type** float

**property expected_range**(*cls*)

Returns the expected range of the objective, which is not necessarily the possible ranges.

For example, our expected R2 range is from [-1, 1], although the actual range is (-inf, 1].

**property greater_is_better**(*cls*)

Returns a boolean determining if a greater score indicates better model performance.

**property is_bounded_like_percentage**(*cls*)

Returns whether this objective is bounded between 0 and 1, inclusive.

**classmethod is_defined_for_problem_type**(*cls*, *problem_type*)

Returns whether or not an objective is defined for a problem type.

**property name**(*cls*)

Returns a name describing the objective.

**abstract classmethod objective_function**(*cls*, *y_true*, *y_predicted*, *X=None*, *sample_weight=None*)

Computes the relative value of the provided predictions compared to the actual labels, according a specified metric.

**Parameters**

- **y_predicted** (*pd.Series*) – Predicted values of length [n_samples]
- **y_true** (*pd.Series*) – Actual class labels of length [n_samples]
- **X** (*pd.DataFrame or np.ndarray*) – Extra data of shape [n_samples, n_features] necessary to calculate score
- **sample_weight** (*pd.DataFrame or np.ndarray*) – Sample weights used in computing objective value result

**Returns** Numerical value used to calculate score

**property perfect_score**(*cls*)

Returns the score obtained by evaluating this objective on a perfect model.

**positive_only**(*cls*)

If True, this objective is only valid for positive data. Defaults to False.

**score**(*self*, *y_true*, *y_predicted*, *X=None*, *sample_weight=None*)

Returns a numerical score indicating performance based on the differences between the predicted and actual values.

**Parameters**

- **y_predicted** (*pd.Series*) – Predicted values of length [n_samples]
- **y_true** (*pd.Series*) – Actual class labels of length [n_samples]
- **X** (*pd.DataFrame or np.ndarray*) – Extra data of shape [n_samples, n_features] necessary to calculate score
- **sample_weight** (*pd.DataFrame or np.ndarray*) – Sample weights used in computing objective value result

**Returns** score

**property score_needs_proba**(*cls*)

Returns a boolean determining if the score() method needs probability estimates.

This should be true for objectives which work with predicted probabilities, like log loss or AUC, and false for objectives which compare predicted class labels to the actual labels, like F1 or correlation.

**validate_inputs**(*self*, *y_true*, *y_predicted*)

Validates the input based on a few simple checks.

**Parameters**

- **y_predicted** (*pd.Series, or pd.DataFrame*) – Predicted values of length [n_samples].
- **y_true** (*pd.Series*) – Actual class labels of length [n_samples].

**Raises** **ValueError** – If the inputs are malformed.

**class** evalml.objectives.**Precision**

Precision score for binary classification.

### Example

```
>>> y_true = pd.Series([0, 1, 1, 0, 0, 1, 1, 1, 0, 1, 1])
>>> y_pred = pd.Series([0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1])
>>> np.testing.assert_almost_equal(Precision().objective_function(y_true, y_pred),␣
↪1.0)
```

**Attributes**

| | |
|---|---|
| **expected_range** | [0, 1] |
| **greater_is_better** | True |
| **is_bounded_like_percentage** | True |
| **name** | Precision |
| **perfect_score** | 1.0 |
| **problem_types** | [ProblemTypes.BINARY, ProblemTypes.TIME_SERIES_BINARY] |
| **score_needs_proba** | False |

**Methods**

| | |
|---|---|
| *calculate_percent_difference* | Calculate the percent difference between scores. |
| *can_optimize_threshold* | Returns a boolean determining if we can optimize the binary classification objective threshold. |
| *decision_function* | Apply a learned threshold to predicted probabilities to get predicted classes. |
| *is_defined_for_problem_type* | Returns whether or not an objective is defined for a problem type. |
| *objective_function* | Objective function for precision score for binary classification. |
| *optimize_threshold* | Learn a binary classification threshold which optimizes the current objective. |
| *positive_only* | If True, this objective is only valid for positive data. Defaults to False. |
| *score* | Returns a numerical score indicating performance based on the differences between the predicted and actual values. |
| *validate_inputs* | Validate inputs for scoring. |

**classmethod calculate_percent_difference**(*cls*, *score*, *baseline_score*)

Calculate the percent difference between scores.

**Parameters**

- **score** (*float*) – A score. Output of the score method of this objective.
- **baseline_score** (*float*) – A score. Output of the score method of this objective. In practice, this is the score achieved on this objective with a baseline estimator.

**Returns**

**The percent difference between the scores. Note that for objectives that can be interpreted** as percentages, this will be the difference between the reference score and score. For all other objectives, the difference will be normalized by the reference score.

**Return type** float

**property can_optimize_threshold**(*cls*)

Returns a boolean determining if we can optimize the binary classification objective threshold.

This will be false for any objective that works directly with predicted probabilities, like log loss and AUC. Otherwise, it will be true.

**Returns** Whether or not an objective can be optimized.

**Return type** bool

**decision_function**(*self*, *ypred_proba*, *threshold=0.5*, *X=None*)

Apply a learned threshold to predicted probabilities to get predicted classes.

**Parameters**

- **ypred_proba** (*pd.Series, np.ndarray*) – The classifier's predicted probabilities
- **threshold** (*float, optional*) – Threshold used to make a prediction. Defaults to 0.5.
- **X** (*pd.DataFrame, optional*) – Any extra columns that are needed from training data.

**Returns** predictions

**classmethod is_defined_for_problem_type**(*cls*, *problem_type*)

Returns whether or not an objective is defined for a problem type.

**objective_function**(*self*, *y_true*, *y_predicted*, *X=None*, *sample_weight=None*)

Objective function for precision score for binary classification.

**optimize_threshold**(*self*, *ypred_proba*, *y_true*, *X=None*)

Learn a binary classification threshold which optimizes the current objective.

**Parameters**

- **ypred_proba** (*pd.Series*) – The classifier's predicted probabilities
- **y_true** (*pd.Series*) – The ground truth for the predictions.
- **X** (*pd.DataFrame, optional*) – Any extra columns that are needed from training data.

**Returns** Optimal threshold for this objective.

**Raises** **RuntimeError** – If objective cannot be optimized.

**positive_only**(*cls*)

If True, this objective is only valid for positive data. Defaults to False.

**score**(*self*, *y_true*, *y_predicted*, *X=None*, *sample_weight=None*)

Returns a numerical score indicating performance based on the differences between the predicted and actual values.

**Parameters**

- **y_predicted** (*pd.Series*) – Predicted values of length [n_samples]
- **y_true** (*pd.Series*) – Actual class labels of length [n_samples]
- **X** (*pd.DataFrame or np.ndarray*) – Extra data of shape [n_samples, n_features] necessary to calculate score
- **sample_weight** (*pd.DataFrame or np.ndarray*) – Sample weights used in computing objective value result

**Returns** score

**validate_inputs**(*self*, *y_true*, *y_predicted*)

Validate inputs for scoring.

**class** evalml.objectives.**PrecisionMacro**

Precision score for multiclass classification using macro-averaging.

### Example

```
>>> y_true = pd.Series([0, 1, 0, 2, 0, 1, 2, 1, 2, 0, 2])
>>> y_pred = pd.Series([0, 0, 0, 0, 1, 1, 1, 1, 2, 2, 2])
>>> np.testing.assert_almost_equal(PrecisionMacro().objective_function(y_true, y_
↪pred), 0.5555555)
```

**Attributes**

| | |
|---|---|
| **expected_range** | [0, 1] |
| **greater_is_better** | True |
| **is_bounded_like_percentage** | True |
| **name** | Precision Macro |
| **perfect_score** | 1.0 |
| **problem_types** | [ProblemTypes.MULTICLASS, ProblemTypes.TIME_SERIES_MULTICLASS] |
| **score_needs_proba** | False |

**Methods**

| | |
|---|---|
| *calculate_percent_difference* | Calculate the percent difference between scores. |
| *is_defined_for_problem_type* | Returns whether or not an objective is defined for a problem type. |
| *objective_function* | Objective function for precision score for multiclass classification using macro-averaging. |
| *positive_only* | If True, this objective is only valid for positive data. Defaults to False. |
| *score* | Returns a numerical score indicating performance based on the differences between the predicted and actual values. |
| *validate_inputs* | Validates the input based on a few simple checks. |

**classmethod calculate_percent_difference**(*cls*, *score*, *baseline_score*)

Calculate the percent difference between scores.

**Parameters**

- **score** (*float*) – A score. Output of the score method of this objective.
- **baseline_score** (*float*) – A score. Output of the score method of this objective. In practice, this is the score achieved on this objective with a baseline estimator.

**Returns**

**The percent difference between the scores. Note that for objectives that can be interpreted** as percentages, this will be the difference between the reference score and score. For all other objectives, the difference will be normalized by the reference score.

**Return type** float

**classmethod is_defined_for_problem_type**(*cls*, *problem_type*)

Returns whether or not an objective is defined for a problem type.

**objective_function**(*self*, *y_true*, *y_predicted*, *X=None*, *sample_weight=None*)

Objective function for precision score for multiclass classification using macro-averaging.

**positive_only**(*cls*)

If True, this objective is only valid for positive data. Defaults to False.

**score**(*self*, *y_true*, *y_predicted*, *X=None*, *sample_weight=None*)

Returns a numerical score indicating performance based on the differences between the predicted and actual values.

**Parameters**

- **y_predicted** (*pd.Series*) – Predicted values of length [n_samples]
- **y_true** (*pd.Series*) – Actual class labels of length [n_samples]
- **X** (*pd.DataFrame or np.ndarray*) – Extra data of shape [n_samples, n_features] necessary to calculate score
- **sample_weight** (*pd.DataFrame or np.ndarray*) – Sample weights used in computing objective value result

**Returns** score

**validate_inputs**(*self*, *y_true*, *y_predicted*)

Validates the input based on a few simple checks.

**Parameters**

- **y_predicted** (*pd.Series, or pd.DataFrame*) – Predicted values of length [n_samples].
- **y_true** (*pd.Series*) – Actual class labels of length [n_samples].

**Raises** **ValueError** – If the inputs are malformed.

**class** evalml.objectives.**PrecisionMicro**

Precision score for multiclass classification using micro averaging.

### Example

```
>>> y_true = pd.Series([0, 1, 0, 2, 0, 1, 2, 1, 2, 0, 2])
>>> y_pred = pd.Series([0, 0, 0, 0, 1, 1, 1, 1, 2, 2, 2])
>>> np.testing.assert_almost_equal(PrecisionMicro().objective_function(y_true, y_
↪pred), 0.5454545)
```

**Attributes**

| | |
|---|---|
| **expected_range** | [0, 1] |
| **greater_is_better** | True |
| **is_bounded_like** | True |
| **name** | Precision Micro |
| **perfect_score** | 1.0 |
| **problem_types** | [ProblemTypes.MULTICLASS, ProblemTypes.TIME_SERIES_MULTICLASS] |
| **score_needs_proba** | False |

**Methods**

| | |
|---|---|
| *calculate_percent_difference* | Calculate the percent difference between scores. |
| *is_defined_for_problem_type* | Returns whether or not an objective is defined for a problem type. |
| *objective_function* | Objective function for precision score for binary classification using micro-averaging. |
| *positive_only* | If True, this objective is only valid for positive data. Defaults to False. |
| *score* | Returns a numerical score indicating performance based on the differences between the predicted and actual values. |
| *validate_inputs* | Validates the input based on a few simple checks. |

**classmethod calculate_percent_difference**(*cls*, *score*, *baseline_score*)

Calculate the percent difference between scores.

**Parameters**

- **score** (*float*) – A score. Output of the score method of this objective.
- **baseline_score** (*float*) – A score. Output of the score method of this objective. In practice, this is the score achieved on this objective with a baseline estimator.

**Returns**

**The percent difference between the scores. Note that for objectives that can be interpreted** as percentages, this will be the difference between the reference score and score. For all other objectives, the difference will be normalized by the reference score.

**Return type** float

**classmethod is_defined_for_problem_type**(*cls*, *problem_type*)

Returns whether or not an objective is defined for a problem type.

**objective_function**(*self*, *y_true*, *y_predicted*, *X=None*, *sample_weight=None*)

Objective function for precision score for binary classification using micro-averaging.

**positive_only**(*cls*)

If True, this objective is only valid for positive data. Defaults to False.

**score**(*self*, *y_true*, *y_predicted*, *X=None*, *sample_weight=None*)

Returns a numerical score indicating performance based on the differences between the predicted and actual values.

**Parameters**

- **y_predicted** (*pd.Series*) – Predicted values of length [n_samples]
- **y_true** (*pd.Series*) – Actual class labels of length [n_samples]
- **X** (*pd.DataFrame or np.ndarray*) – Extra data of shape [n_samples, n_features] necessary to calculate score
- **sample_weight** (*pd.DataFrame or np.ndarray*) – Sample weights used in computing objective value result

**Returns** score

**validate_inputs**(*self*, *y_true*, *y_predicted*)

Validates the input based on a few simple checks.

**Parameters**

- **y_predicted** (*pd.Series, or pd.DataFrame*) – Predicted values of length [n_samples].
- **y_true** (*pd.Series*) – Actual class labels of length [n_samples].

**Raises** **ValueError** – If the inputs are malformed.

**class** evalml.objectives.**PrecisionWeighted**

Precision score for multiclass classification using weighted averaging.

### Example

```
>>> y_true = pd.Series([0, 1, 0, 2, 0, 1, 2, 1, 2, 0, 2])
>>> y_pred = pd.Series([0, 0, 0, 0, 1, 1, 1, 1, 2, 2, 2])
>>> np.testing.assert_almost_equal(PrecisionWeighted().objective_function(y_true, y_
↪pred), 0.5606060)
```

**Attributes**

| | |
|---|---|
| **expected_range** | [0, 1] |
| **greater_is_better** | True |
| **is_bounded_like** | True |
| **name** | Precision Weighted |
| **perfect_score** | 1.0 |
| **problem_types** | [ProblemTypes.MULTICLASS, ProblemTypes.TIME_SERIES_MULTICLASS] |
| **score_needs_proba** | False |

**Methods**

| | |
|---|---|
| *calculate_percent_difference* | Calculate the percent difference between scores. |
| *is_defined_for_problem_type* | Returns whether or not an objective is defined for a problem type. |
| *objective_function* | Objective function for precision score for multiclass classification using weighted averaging. |
| *positive_only* | If True, this objective is only valid for positive data. Defaults to False. |
| *score* | Returns a numerical score indicating performance based on the differences between the predicted and actual values. |
| *validate_inputs* | Validates the input based on a few simple checks. |

**classmethod calculate_percent_difference**(*cls*, *score*, *baseline_score*)

Calculate the percent difference between scores.

**Parameters**

- **score** (*float*) – A score. Output of the score method of this objective.
- **baseline_score** (*float*) – A score. Output of the score method of this objective. In practice, this is the score achieved on this objective with a baseline estimator.

**Returns**

**The percent difference between the scores. Note that for objectives that can be interpreted** as percentages, this will be the difference between the reference score and score. For all other objectives, the difference will be normalized by the reference score.

**Return type** float

**classmethod is_defined_for_problem_type**(*cls*, *problem_type*)

Returns whether or not an objective is defined for a problem type.

**objective_function**(*self*, *y_true*, *y_predicted*, *X=None*, *sample_weight=None*)

Objective function for precision score for multiclass classification using weighted averaging.

**positive_only**(*cls*)

If True, this objective is only valid for positive data. Defaults to False.

**score**(*self*, *y_true*, *y_predicted*, *X=None*, *sample_weight=None*)

Returns a numerical score indicating performance based on the differences between the predicted and actual values.

**Parameters**

- **y_predicted** (*pd.Series*) – Predicted values of length [n_samples]
- **y_true** (*pd.Series*) – Actual class labels of length [n_samples]
- **X** (*pd.DataFrame or np.ndarray*) – Extra data of shape [n_samples, n_features] necessary to calculate score
- **sample_weight** (*pd.DataFrame or np.ndarray*) – Sample weights used in computing objective value result

**Returns** score

**validate_inputs**(*self*, *y_true*, *y_predicted*)

Validates the input based on a few simple checks.

**Parameters**

- **y_predicted** (*pd.Series, or pd.DataFrame*) – Predicted values of length [n_samples].
- **y_true** (*pd.Series*) – Actual class labels of length [n_samples].

**Raises** **ValueError** – If the inputs are malformed.

**class** evalml.objectives.**R2**

Coefficient of determination for regression.

### Example

```
>>> y_true = pd.Series([1.5, 2, 3, 1, 0.5, 1, 2.5, 2.5, 1, 0.5, 2])
>>> y_pred = pd.Series([1.5, 2.5, 2, 1, 0.5, 1, 3, 2.25, 0.75, 0.25, 1.75])
>>> np.testing.assert_almost_equal(R2().objective_function(y_true, y_pred), 0.
↪7638036)
```

**Attributes**

| | |
|---|---|
| **expected_range** | None |
| **greater_is_better** | True |
| **is_bounded_like_percentage** | False |
| **name** | R2 |
| **perfect_score** | 1 |
| **problem_types** | [ProblemTypes.REGRESSION, ProblemTypes.TIME_SERIES_REGRESSION] |
| **score_needs_proba** | False |

**Methods**

| | |
|---|---|
| *calculate_percent_difference* | Calculate the percent difference between scores. |
| *is_defined_for_problem_type* | Returns whether or not an objective is defined for a problem type. |
| *objective_function* | Objective function for coefficient of determination for regression. |
| *positive_only* | If True, this objective is only valid for positive data. Defaults to False. |
| *score* | Returns a numerical score indicating performance based on the differences between the predicted and actual values. |
| *validate_inputs* | Validates the input based on a few simple checks. |

**classmethod calculate_percent_difference**(*cls*, *score*, *baseline_score*)

Calculate the percent difference between scores.

**Parameters**

- **score** (*float*) – A score. Output of the score method of this objective.
- **baseline_score** (*float*) – A score. Output of the score method of this objective. In practice, this is the score achieved on this objective with a baseline estimator.

**Returns**

**The percent difference between the scores. Note that for objectives that can be interpreted** as percentages, this will be the difference between the reference score and score. For all other objectives, the difference will be normalized by the reference score.

**Return type** float

`classmethod is_defined_for_problem_type`(*cls*, *problem_type*)

Returns whether or not an objective is defined for a problem type.

`objective_function`(*self*, *y_true*, *y_predicted*, *X=None*, *sample_weight=None*)

Objective function for coefficient of determination for regression.

`positive_only`(*cls*)

If True, this objective is only valid for positive data. Defaults to False.

`score`(*self*, *y_true*, *y_predicted*, *X=None*, *sample_weight=None*)

Returns a numerical score indicating performance based on the differences between the predicted and actual values.

**Parameters**

- **y_predicted** (*pd.Series*) – Predicted values of length [n_samples]
- **y_true** (*pd.Series*) – Actual class labels of length [n_samples]
- **X** (*pd.DataFrame or np.ndarray*) – Extra data of shape [n_samples, n_features] necessary to calculate score
- **sample_weight** (*pd.DataFrame or np.ndarray*) – Sample weights used in computing objective value result

**Returns** score

`validate_inputs`(*self*, *y_true*, *y_predicted*)

Validates the input based on a few simple checks.

**Parameters**

- **y_predicted** (*pd.Series, or pd.DataFrame*) – Predicted values of length [n_samples].
- **y_true** (*pd.Series*) – Actual class labels of length [n_samples].

**Raises** **ValueError** – If the inputs are malformed.

`evalml.objectives.ranking_only_objectives()`

Get ranking-only objective classes.

Ranking-only objectives are objectives that are useful for evaluating the performance of a model, but should not be used as an optimization objective during AutoMLSearch for various reasons.

**Returns** List of ObjectiveBase classes

**class** `evalml.objectives.Recall`

Recall score for binary classification.

### Example

```
>>> y_true = pd.Series([0, 1, 1, 0, 0, 1, 1, 1, 0, 1, 1])
>>> y_pred = pd.Series([0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1])
>>> np.testing.assert_almost_equal(Recall().objective_function(y_true, y_pred), 0.
↪1428571)
```

**Attributes**

| | |
|---|---|
| **expected_range** | [0, 1] |
| **greater_is_better** | True |
| **is_bounded_like_percentage** | True |
| **name** | Recall |
| **perfect_score** | 1.0 |
| **problem_types** | [ProblemTypes.BINARY, ProblemTypes.TIME_SERIES_BINARY] |
| **score_needs_proba** | False |

**Methods**

| | |
|---|---|
| *calculate_percent_difference* | Calculate the percent difference between scores. |
| *can_optimize_threshold* | Returns a boolean determining if we can optimize the binary classification objective threshold. |
| *decision_function* | Apply a learned threshold to predicted probabilities to get predicted classes. |
| *is_defined_for_problem_type* | Returns whether or not an objective is defined for a problem type. |
| *objective_function* | Objective function for recall score for binary classification. |
| *optimize_threshold* | Learn a binary classification threshold which optimizes the current objective. |
| *positive_only* | If True, this objective is only valid for positive data. Defaults to False. |
| *score* | Returns a numerical score indicating performance based on the differences between the predicted and actual values. |
| *validate_inputs* | Validate inputs for scoring. |

**classmethod calculate_percent_difference**(*cls*, *score*, *baseline_score*)

Calculate the percent difference between scores.

**Parameters**

- **score** (*float*) – A score. Output of the score method of this objective.
- **baseline_score** (*float*) – A score. Output of the score method of this objective. In practice, this is the score achieved on this objective with a baseline estimator.

**Returns**

**The percent difference between the scores. Note that for objectives that can be interpreted** as percentages, this will be the difference between the reference score and score. For all other objectives, the difference will be normalized by the reference score.

**Return type** float

**property can_optimize_threshold**(*cls*)

Returns a boolean determining if we can optimize the binary classification objective threshold.

This will be false for any objective that works directly with predicted probabilities, like log loss and AUC. Otherwise, it will be true.

**Returns** Whether or not an objective can be optimized.

**Return type** bool

**decision_function**(*self*, *ypred_proba*, *threshold=0.5*, *X=None*)

Apply a learned threshold to predicted probabilities to get predicted classes.

**Parameters**

- **ypred_proba** (*pd.Series, np.ndarray*) – The classifier's predicted probabilities
- **threshold** (*float, optional*) – Threshold used to make a prediction. Defaults to 0.5.
- **X** (*pd.DataFrame, optional*) – Any extra columns that are needed from training data.

**Returns** predictions

**classmethod is_defined_for_problem_type**(*cls*, *problem_type*)

Returns whether or not an objective is defined for a problem type.

**objective_function**(*self*, *y_true*, *y_predicted*, *X=None*, *sample_weight=None*)

Objective function for recall score for binary classification.

**optimize_threshold**(*self*, *ypred_proba*, *y_true*, *X=None*)

Learn a binary classification threshold which optimizes the current objective.

**Parameters**

- **ypred_proba** (*pd.Series*) – The classifier's predicted probabilities
- **y_true** (*pd.Series*) – The ground truth for the predictions.
- **X** (*pd.DataFrame, optional*) – Any extra columns that are needed from training data.

**Returns** Optimal threshold for this objective.

**Raises** **RuntimeError** – If objective cannot be optimized.

**positive_only**(*cls*)

If True, this objective is only valid for positive data. Defaults to False.

**score**(*self*, *y_true*, *y_predicted*, *X=None*, *sample_weight=None*)

Returns a numerical score indicating performance based on the differences between the predicted and actual values.

**Parameters**

- **y_predicted** (*pd.Series*) – Predicted values of length [n_samples]
- **y_true** (*pd.Series*) – Actual class labels of length [n_samples]
- **X** (*pd.DataFrame or np.ndarray*) – Extra data of shape [n_samples, n_features] necessary to calculate score
- **sample_weight** (*pd.DataFrame or np.ndarray*) – Sample weights used in computing objective value result

**Returns** score

**validate_inputs**(*self*, *y_true*, *y_predicted*)

Validate inputs for scoring.

**class** evalml.objectives.**RecallMacro**

Recall score for multiclass classification using macro averaging.

### Example

```
>>> y_true = pd.Series([0, 1, 0, 2, 0, 1, 2, 1, 2, 0, 2])
>>> y_pred = pd.Series([0, 0, 0, 0, 1, 1, 1, 1, 2, 2, 2])
>>> np.testing.assert_almost_equal(RecallMacro().objective_function(y_true, y_pred),
↪ 0.5555555)
```

**Attributes**

| | |
|---|---|
| **expected_range** | [0, 1] |
| **greater_is_better** | True |
| **is_bounded_like_percentage** | True |
| **name** | Recall Macro |
| **perfect_score** | 1.0 |
| **problem_types** | [ProblemTypes.MULTICLASS, ProblemTypes.TIME_SERIES_MULTICLASS] |
| **score_needs_proba** | False |

**Methods**

| | |
|---|---|
| *calculate_percent_difference* | Calculate the percent difference between scores. |
| *is_defined_for_problem_type* | Returns whether or not an objective is defined for a problem type. |
| *objective_function* | Objective function for recall score for multiclass classification using macro-averaging. |
| *positive_only* | If True, this objective is only valid for positive data. Defaults to False. |
| *score* | Returns a numerical score indicating performance based on the differences between the predicted and actual values. |
| *validate_inputs* | Validates the input based on a few simple checks. |

**classmethod calculate_percent_difference**(*cls*, *score*, *baseline_score*)

Calculate the percent difference between scores.

**Parameters**

- **score** (*float*) – A score. Output of the score method of this objective.
- **baseline_score** (*float*) – A score. Output of the score method of this objective. In practice, this is the score achieved on this objective with a baseline estimator.

**Returns**

**The percent difference between the scores. Note that for objectives that can be interpreted** as percentages, this will be the difference between the reference score and score. For all other objectives, the difference will be normalized by the reference score.

**Return type** float

`classmethod is_defined_for_problem_type`(*cls*, *problem_type*)

Returns whether or not an objective is defined for a problem type.

`objective_function`(*self*, *y_true*, *y_predicted*, *X=None*, *sample_weight=None*)

Objective function for recall score for multiclass classification using macro-averaging.

`positive_only`(*cls*)

If True, this objective is only valid for positive data. Defaults to False.

`score`(*self*, *y_true*, *y_predicted*, *X=None*, *sample_weight=None*)

Returns a numerical score indicating performance based on the differences between the predicted and actual values.

**Parameters**

- **y_predicted** (*pd.Series*) – Predicted values of length [n_samples]
- **y_true** (*pd.Series*) – Actual class labels of length [n_samples]
- **X** (*pd.DataFrame or np.ndarray*) – Extra data of shape [n_samples, n_features] necessary to calculate score
- **sample_weight** (*pd.DataFrame or np.ndarray*) – Sample weights used in computing objective value result

**Returns** score

`validate_inputs`(*self*, *y_true*, *y_predicted*)

Validates the input based on a few simple checks.

**Parameters**

- **y_predicted** (*pd.Series, or pd.DataFrame*) – Predicted values of length [n_samples].
- **y_true** (*pd.Series*) – Actual class labels of length [n_samples].

**Raises** **ValueError** – If the inputs are malformed.

**class** `evalml.objectives.`**RecallMicro**

Recall score for multiclass classification using micro averaging.

### Example

```
>>> y_true = pd.Series([0, 1, 0, 2, 0, 1, 2, 1, 2, 0, 2])
>>> y_pred = pd.Series([0, 0, 0, 0, 1, 1, 1, 1, 2, 2, 2])
>>> np.testing.assert_almost_equal(RecallMicro().objective_function(y_true, y_pred),
↪ 0.5454545)
```

**Attributes**

| | |
|---|---|
| **expected_range** | [0, 1] |
| **greater_is_better** | True |
| **is_bounded_like_percentage** | True |
| **name** | Recall Micro |
| **perfect_score** | 1.0 |
| **problem_types** | [ProblemTypes.MULTICLASS, ProblemTypes.TIME_SERIES_MULTICLASS] |
| **score_needs_proba** | False |

**Methods**

| | |
|---|---|
| *calculate_percent_difference* | Calculate the percent difference between scores. |
| *is_defined_for_problem_type* | Returns whether or not an objective is defined for a problem type. |
| *objective_function* | Objective function for recall score for multiclass classification using micro-averaging. |
| *positive_only* | If True, this objective is only valid for positive data. Defaults to False. |
| *score* | Returns a numerical score indicating performance based on the differences between the predicted and actual values. |
| *validate_inputs* | Validates the input based on a few simple checks. |

**classmethod calculate_percent_difference**(*cls*, *score*, *baseline_score*)

Calculate the percent difference between scores.

**Parameters**

- **score** (*float*) – A score. Output of the score method of this objective.
- **baseline_score** (*float*) – A score. Output of the score method of this objective. In practice, this is the score achieved on this objective with a baseline estimator.

**Returns**

**The percent difference between the scores. Note that for objectives that can be interpreted** as percentages, this will be the difference between the reference score and score. For all other objectives, the difference will be normalized by the reference score.

**Return type** float

**classmethod is_defined_for_problem_type**(*cls*, *problem_type*)

Returns whether or not an objective is defined for a problem type.

**objective_function**(*self*, *y_true*, *y_predicted*, *X=None*, *sample_weight=None*)

Objective function for recall score for multiclass classification using micro-averaging.

**positive_only**(*cls*)

If True, this objective is only valid for positive data. Defaults to False.

**score**(*self*, *y_true*, *y_predicted*, *X=None*, *sample_weight=None*)

Returns a numerical score indicating performance based on the differences between the predicted and actual values.

**Parameters**

- **y_predicted** (*pd.Series*) – Predicted values of length [n_samples]
- **y_true** (*pd.Series*) – Actual class labels of length [n_samples]
- **X** (*pd.DataFrame or np.ndarray*) – Extra data of shape [n_samples, n_features] necessary to calculate score
- **sample_weight** (*pd.DataFrame or np.ndarray*) – Sample weights used in computing objective value result

**Returns** score

**validate_inputs**(*self*, *y_true*, *y_predicted*)

Validates the input based on a few simple checks.

**Parameters**

- **y_predicted** (*pd.Series, or pd.DataFrame*) – Predicted values of length [n_samples].
- **y_true** (*pd.Series*) – Actual class labels of length [n_samples].

**Raises** **ValueError** – If the inputs are malformed.

**class** evalml.objectives.**RecallWeighted**

Recall score for multiclass classification using weighted averaging.

### Example

```
>>> y_true = pd.Series([0, 1, 0, 2, 0, 1, 2, 1, 2, 0, 2])
>>> y_pred = pd.Series([0, 0, 0, 0, 1, 1, 1, 1, 2, 2, 2])
>>> np.testing.assert_almost_equal(RecallWeighted().objective_function(y_true, y_
↪pred), 0.5454545)
```

**Attributes**

| | |
|---|---|
| **expected_range** | [0, 1] |
| **greater_is_better** | True |
| **is_bounded_like** | True |
| **name** | Recall Weighted |
| **perfect_score** | 1.0 |
| **problem_types** | [ProblemTypes.MULTICLASS, ProblemTypes.TIME_SERIES_MULTICLASS] |
| **score_needs_proba** | False |

**Methods**

| | |
|---|---|
| *calculate_percent_difference* | Calculate the percent difference between scores. |
| *is_defined_for_problem_type* | Returns whether or not an objective is defined for a problem type. |
| *objective_function* | Objective function for recall score for multiclass classification using weighted averaging. |
| *positive_only* | If True, this objective is only valid for positive data. Defaults to False. |
| *score* | Returns a numerical score indicating performance based on the differences between the predicted and actual values. |
| *validate_inputs* | Validates the input based on a few simple checks. |

**classmethod calculate_percent_difference**(*cls*, *score*, *baseline_score*)

Calculate the percent difference between scores.

**Parameters**

- **score** (*float*) – A score. Output of the score method of this objective.
- **baseline_score** (*float*) – A score. Output of the score method of this objective. In practice, this is the score achieved on this objective with a baseline estimator.

**Returns**

**The percent difference between the scores. Note that for objectives that can be interpreted** as percentages, this will be the difference between the reference score and score. For all other objectives, the difference will be normalized by the reference score.

**Return type** float

**classmethod is_defined_for_problem_type**(*cls*, *problem_type*)

Returns whether or not an objective is defined for a problem type.

**objective_function**(*self*, *y_true*, *y_predicted*, *X=None*, *sample_weight=None*)

Objective function for recall score for multiclass classification using weighted averaging.

**positive_only**(*cls*)

If True, this objective is only valid for positive data. Defaults to False.

**score**(*self*, *y_true*, *y_predicted*, *X=None*, *sample_weight=None*)

Returns a numerical score indicating performance based on the differences between the predicted and actual values.

**Parameters**

- **y_predicted** (*pd.Series*) – Predicted values of length [n_samples]
- **y_true** (*pd.Series*) – Actual class labels of length [n_samples]
- **X** (*pd.DataFrame or np.ndarray*) – Extra data of shape [n_samples, n_features] necessary to calculate score
- **sample_weight** (*pd.DataFrame or np.ndarray*) – Sample weights used in computing objective value result

**Returns** score

**validate_inputs**(*self*, *y_true*, *y_predicted*)

Validates the input based on a few simple checks.

**Parameters**

- **y_predicted** (*pd.Series, or pd.DataFrame*) – Predicted values of length [n_samples].
- **y_true** (*pd.Series*) – Actual class labels of length [n_samples].

**Raises** **ValueError** – If the inputs are malformed.

**class** evalml.objectives.**RegressionObjective**

Base class for all regression objectives.

**Attributes**

| | |
|---|---|
| **problem_types** | [ProblemTypes.REGRESSION, ProblemTypes.TIME_SERIES_REGRESSION] |

**Methods**

| | |
|---|---|
| *calculate_percent_difference* | Calculate the percent difference between scores. |
| *expected_range* | Returns the expected range of the objective, which is not necessarily the possible ranges. |
| *greater_is_better* | Returns a boolean determining if a greater score indicates better model performance. |
| *is_bounded_like_percentage* | Returns whether this objective is bounded between 0 and 1, inclusive. |
| *is_defined_for_problem_type* | Returns whether or not an objective is defined for a problem type. |
| *name* | Returns a name describing the objective. |
| *objective_function* | Computes the relative value of the provided predictions compared to the actual labels, according a specified metric. |
| *perfect_score* | Returns the score obtained by evaluating this objective on a perfect model. |
| *positive_only* | If True, this objective is only valid for positive data. Defaults to False. |
| *score* | Returns a numerical score indicating performance based on the differences between the predicted and actual values. |
| *score_needs_proba* | Returns a boolean determining if the score() method needs probability estimates. |
| *validate_inputs* | Validates the input based on a few simple checks. |

**classmethod calculate_percent_difference**(*cls*, *score*, *baseline_score*)

Calculate the percent difference between scores.

**Parameters**

- **score** (*float*) – A score. Output of the score method of this objective.
- **baseline_score** (*float*) – A score. Output of the score method of this objective. In practice, this is the score achieved on this objective with a baseline estimator.

**Returns**

**The percent difference between the scores. Note that for objectives that can be interpreted** as percentages, this will be the difference between the reference score and score. For all other objectives, the difference will be normalized by the reference score.

**Return type** float

**property expected_range**(*cls*)

Returns the expected range of the objective, which is not necessarily the possible ranges.

For example, our expected R2 range is from [-1, 1], although the actual range is (-inf, 1].

**property greater_is_better**(*cls*)

Returns a boolean determining if a greater score indicates better model performance.

**property is_bounded_like_percentage**(*cls*)

Returns whether this objective is bounded between 0 and 1, inclusive.

**classmethod is_defined_for_problem_type**(*cls*, *problem_type*)

Returns whether or not an objective is defined for a problem type.

**property name**(*cls*)

Returns a name describing the objective.

**abstract classmethod objective_function**(*cls*, *y_true*, *y_predicted*, *X=None*, *sample_weight=None*)

Computes the relative value of the provided predictions compared to the actual labels, according a specified metric.

**Parameters**

- **y_predicted** (*pd.Series*) – Predicted values of length [n_samples]
- **y_true** (*pd.Series*) – Actual class labels of length [n_samples]
- **X** (*pd.DataFrame or np.ndarray*) – Extra data of shape [n_samples, n_features] necessary to calculate score
- **sample_weight** (*pd.DataFrame or np.ndarray*) – Sample weights used in computing objective value result

**Returns** Numerical value used to calculate score

**property perfect_score**(*cls*)

Returns the score obtained by evaluating this objective on a perfect model.

**positive_only**(*cls*)

If True, this objective is only valid for positive data. Defaults to False.

**score**(*self*, *y_true*, *y_predicted*, *X=None*, *sample_weight=None*)

Returns a numerical score indicating performance based on the differences between the predicted and actual values.

**Parameters**

- **y_predicted** (*pd.Series*) – Predicted values of length [n_samples]
- **y_true** (*pd.Series*) – Actual class labels of length [n_samples]
- **X** (*pd.DataFrame or np.ndarray*) – Extra data of shape [n_samples, n_features] necessary to calculate score
- **sample_weight** (*pd.DataFrame or np.ndarray*) – Sample weights used in computing objective value result

**Returns** score

**property score_needs_proba**(*cls*)

Returns a boolean determining if the score() method needs probability estimates.

This should be true for objectives which work with predicted probabilities, like log loss or AUC, and false for objectives which compare predicted class labels to the actual labels, like F1 or correlation.

**validate_inputs**(*self*, *y_true*, *y_predicted*)

Validates the input based on a few simple checks.

**Parameters**

- **y_predicted** (*pd.Series, or pd.DataFrame*) – Predicted values of length [n_samples].
- **y_true** (*pd.Series*) – Actual class labels of length [n_samples].

**Raises** **ValueError** – If the inputs are malformed.

**class** evalml.objectives.**RootMeanSquaredError**

Root mean squared error for regression.

### Example

```
>>> y_true = pd.Series([1.5, 2, 3, 1, 0.5, 1, 2.5, 2.5, 1, 0.5, 2])
>>> y_pred = pd.Series([1.5, 2.5, 2, 1, 0.5, 1, 3, 2.25, 0.75, 0.25, 1.75])
>>> np.testing.assert_almost_equal(RootMeanSquaredError().objective_function(y_true,
↪ y_pred), 0.3988620)
```

**Attributes**

| | |
|---|---|
| **expected_range** | None |
| **greater_is_better** | False |
| **is_bounded_like_percentage** | False |
| **name** | Root Mean Squared Error |
| **perfect_score** | 0.0 |
| **problem_types** | [ProblemTypes.REGRESSION, ProblemTypes.TIME_SERIES_REGRESSION] |
| **score_needs_proba** | False |

**Methods**

| | |
|---|---|
| *calculate_percent_difference* | Calculate the percent difference between scores. |
| *is_defined_for_problem_type* | Returns whether or not an objective is defined for a problem type. |
| *objective_function* | Objective function for root mean squared error for regression. |
| *positive_only* | If True, this objective is only valid for positive data. Defaults to False. |
| *score* | Returns a numerical score indicating performance based on the differences between the predicted and actual values. |
| *validate_inputs* | Validates the input based on a few simple checks. |

**classmethod calculate_percent_difference**(*cls*, *score*, *baseline_score*)

Calculate the percent difference between scores.

**Parameters**

- **score** (*float*) – A score. Output of the score method of this objective.
- **baseline_score** (*float*) – A score. Output of the score method of this objective. In practice, this is the score achieved on this objective with a baseline estimator.

**Returns**

**The percent difference between the scores. Note that for objectives that can be interpreted** as percentages, this will be the difference between the reference score and score. For all other objectives, the difference will be normalized by the reference score.

**Return type** float

**classmethod is_defined_for_problem_type**(*cls*, *problem_type*)

Returns whether or not an objective is defined for a problem type.

**objective_function**(*self*, *y_true*, *y_predicted*, *X=None*, *sample_weight=None*)

Objective function for root mean squared error for regression.

**positive_only**(*cls*)

If True, this objective is only valid for positive data. Defaults to False.

**score**(*self*, *y_true*, *y_predicted*, *X=None*, *sample_weight=None*)

Returns a numerical score indicating performance based on the differences between the predicted and actual values.

**Parameters**

- **y_predicted** (*pd.Series*) – Predicted values of length [n_samples]
- **y_true** (*pd.Series*) – Actual class labels of length [n_samples]
- **X** (*pd.DataFrame or np.ndarray*) – Extra data of shape [n_samples, n_features] necessary to calculate score
- **sample_weight** (*pd.DataFrame or np.ndarray*) – Sample weights used in computing objective value result

**Returns** score

**validate_inputs**(*self*, *y_true*, *y_predicted*)

Validates the input based on a few simple checks.

**Parameters**

- **y_predicted** (*pd.Series, or pd.DataFrame*) – Predicted values of length [n_samples].
- **y_true** (*pd.Series*) – Actual class labels of length [n_samples].

**Raises** **ValueError** – If the inputs are malformed.

**class** evalml.objectives.**RootMeanSquaredLogError**

Root mean squared log error for regression.

Only valid for nonnegative inputs. Otherwise, will throw a ValueError.

### Example

```
>>> y_true = pd.Series([1.5, 2, 3, 1, 0.5, 1, 2.5, 2.5, 1, 0.5, 2])
>>> y_pred = pd.Series([1.5, 2.5, 2, 1, 0.5, 1, 3, 2.25, 0.75, 0.25, 1.75])
>>> np.testing.assert_almost_equal(RootMeanSquaredLogError().objective_function(y_
↪true, y_pred), 0.13090204)
```

**Attributes**

| | |
|---|---|
| **expected_range** | None |
| **greater_is_better** | False |
| **is_bounded_like_percentage** | False |
| **name** | Root Mean Squared Log Error |
| **perfect_score** | 0.0 |
| **problem_types** | [ProblemTypes.REGRESSION, ProblemTypes.TIME_SERIES_REGRESSION] |
| **score_needs_proba** | False |

**Methods**

| | |
|---|---|
| *calculate_percent_difference* | Calculate the percent difference between scores. |
| *is_defined_for_problem_type* | Returns whether or not an objective is defined for a problem type. |
| *objective_function* | Objective function for root mean squared log error for regression. |
| *positive_only* | If True, this objective is only valid for positive data. |
| *score* | Returns a numerical score indicating performance based on the differences between the predicted and actual values. |
| *validate_inputs* | Validates the input based on a few simple checks. |

**classmethod calculate_percent_difference**(*cls*, *score*, *baseline_score*)

Calculate the percent difference between scores.

**Parameters**

- **score** (*float*) – A score. Output of the score method of this objective.
- **baseline_score** (*float*) – A score. Output of the score method of this objective. In practice, this is the score achieved on this objective with a baseline estimator.

**Returns**

**The percent difference between the scores. Note that for objectives that can be interpreted** as percentages, this will be the difference between the reference score and score. For all other objectives, the difference will be normalized by the reference score.

**Return type** float

**classmethod is_defined_for_problem_type**(*cls*, *problem_type*)

Returns whether or not an objective is defined for a problem type.

**objective_function**(*self*, *y_true*, *y_predicted*, *X=None*, *sample_weight=None*)

Objective function for root mean squared log error for regression.

**positive_only**(*self*)

If True, this objective is only valid for positive data.

**score**(*self*, *y_true*, *y_predicted*, *X=None*, *sample_weight=None*)

Returns a numerical score indicating performance based on the differences between the predicted and actual values.

**Parameters**

- **y_predicted** (*pd.Series*) – Predicted values of length [n_samples]
- **y_true** (*pd.Series*) – Actual class labels of length [n_samples]
- **X** (*pd.DataFrame or np.ndarray*) – Extra data of shape [n_samples, n_features] necessary to calculate score
- **sample_weight** (*pd.DataFrame or np.ndarray*) – Sample weights used in computing objective value result

**Returns** score

**validate_inputs**(*self*, *y_true*, *y_predicted*)

Validates the input based on a few simple checks.

**Parameters**

- **y_predicted** (*pd.Series, or pd.DataFrame*) – Predicted values of length [n_samples].
- **y_true** (*pd.Series*) – Actual class labels of length [n_samples].

**Raises** **ValueError** – If the inputs are malformed.

**class** evalml.objectives.**SensitivityLowAlert**(*alert_rate=0.01*)

Create instance of SensitivityLowAlert.

**Parameters** **alert_rate** (*float*) – percentage of top scores to classify as high risk.

**Attributes**

| | |
|---|---|
| **expected_range** | [0, 1] |
| **greater_is_better** | True |
| **is_bounded_like_percentage** | True |
| **name** | Sensitivity at Low Alert Rates |
| **perfect_score** | 1.0 |
| **problem_types** | [ProblemTypes.BINARY, ProblemTypes.TIME_SERIES_BINARY] |
| **score_needs_proba** | False |

**Methods**

| | |
|---|---|
| *calculate_percent_difference* | Calculate the percent difference between scores. |
| *can_optimize_threshold* | Returns a boolean determining if we can optimize the binary classification objective threshold. |
| *decision_function* | Determine if an observation is high risk given an alert rate. |
| *is_defined_for_problem_type* | Returns whether or not an objective is defined for a problem type. |
| *objective_function* | Calculate sensitivity across all predictions, using the top alert_rate percent of observations as the predicted positive class. |
| *optimize_threshold* | Learn a binary classification threshold which optimizes the current objective. |
| *positive_only* | If True, this objective is only valid for positive data. Defaults to False. |
| *score* | Returns a numerical score indicating performance based on the differences between the predicted and actual values. |
| *validate_inputs* | Validate inputs for scoring. |

**classmethod calculate_percent_difference**(*cls*, *score*, *baseline_score*)

Calculate the percent difference between scores.

**Parameters**

- **score** (*float*) – A score. Output of the score method of this objective.
- **baseline_score** (*float*) – A score. Output of the score method of this objective. In practice, this is the score achieved on this objective with a baseline estimator.

**Returns**

**The percent difference between the scores. Note that for objectives that can be interpreted** as percentages, this will be the difference between the reference score and score. For all other objectives, the difference will be normalized by the reference score.

**Return type** float

**property can_optimize_threshold**(*cls*)

Returns a boolean determining if we can optimize the binary classification objective threshold.

This will be false for any objective that works directly with predicted probabilities, like log loss and AUC. Otherwise, it will be true.

**Returns** Whether or not an objective can be optimized.

**Return type** bool

**decision_function**(*self*, *ypred_proba*, ***kwargs*)

Determine if an observation is high risk given an alert rate.

**Parameters**

- **ypred_proba** (*pd.Series*) – Predicted probabilities.
- ****kwargs** – Additional abritrary parameters.

**Returns** Whether or not an observation is high risk given an alert rate.

**Return type** pd.Series

**classmethod is_defined_for_problem_type**(*cls*, *problem_type*)

Returns whether or not an objective is defined for a problem type.

**objective_function**(*self*, *y_true*, *y_predicted*, ***kwargs*)

Calculate sensitivity across all predictions, using the top alert_rate percent of observations as the predicted positive class.

**Parameters**

- **y_true** (*pd.Series*) – True labels.
- **y_predicted** (*pd.Series*) – Predicted labels based on alert_rate.
- ****kwargs** – Additional abritrary parameters.

**Returns** sensitivity using the observations with the top scores as the predicted positive class.

**Return type** float

**optimize_threshold**(*self*, *ypred_proba*, *y_true*, *X=None*)

Learn a binary classification threshold which optimizes the current objective.

**Parameters**

- **ypred_proba** (*pd.Series*) – The classifier's predicted probabilities
- **y_true** (*pd.Series*) – The ground truth for the predictions.
- **X** (*pd.DataFrame, optional*) – Any extra columns that are needed from training data.

**Returns** Optimal threshold for this objective.

**Raises** **RuntimeError** – If objective cannot be optimized.

**positive_only**(*cls*)

If True, this objective is only valid for positive data. Defaults to False.

**score**(*self*, *y_true*, *y_predicted*, *X=None*, *sample_weight=None*)

Returns a numerical score indicating performance based on the differences between the predicted and actual values.

**Parameters**

- **y_predicted** (*pd.Series*) – Predicted values of length [n_samples]
- **y_true** (*pd.Series*) – Actual class labels of length [n_samples]
- **X** (*pd.DataFrame or np.ndarray*) – Extra data of shape [n_samples, n_features] necessary to calculate score
- **sample_weight** (*pd.DataFrame or np.ndarray*) – Sample weights used in computing objective value result

**Returns** score

**validate_inputs**(*self*, *y_true*, *y_predicted*)

Validate inputs for scoring.

### Pipelines

EvalML pipelines.

### Subpackages

### components

EvalML component classes.

### Subpackages

### ensemble

Ensemble components.

### Submodules

### stacked_ensemble_base

Stacked Ensemble Base.

### Module Contents

### Classes Summary

| | |
|---|---|
| *StackedEnsembleBase* | Stacked Ensemble Base Class. |

### Contents

**class** evalml.pipelines.components.ensemble.stacked_ensemble_base.**StackedEnsembleBase**(*final_estimator=None, n_jobs=-1, random_seed=0, **kwargs*)

Stacked Ensemble Base Class.

**Parameters**

- **final_estimator** (*Estimator or subclass*) – The estimator used to combine the base estimators.
- **n_jobs** (*int or None*) – Integer describing level of parallelism used for pipelines. None and 1 are equivalent. If set to -1, all CPUs are used. For n_jobs greater than -1, (n_cpus + 1 + n_jobs) are used. Defaults to -1. - Note: there could be some multi-process errors thrown for values of *n_jobs != 1*. If this is the case, please use *n_jobs = 1*.

- **random_seed** (*int*) – Seed for the random number generator. Defaults to 0.

**Attributes**

| | |
|---|---|
| **model_family** | ModelFamily.ENSEMBLE |
| **modifies_features** | True |
| **modifies_target** | False |
| **training_only** | False |

**Methods**

| | |
|---|---|
| *clone* | Constructs a new component with the same parameters and random state. |
| *default_parameters* | Returns the default parameters for stacked ensemble classes. |
| *describe* | Describe a component and its parameters. |
| *feature_importance* | Not implemented for StackedEnsembleClassifier and StackedEnsembleRegressor. |
| *fit* | Fits estimator to data. |
| *get_prediction_intervals* | Find the prediction intervals using the fitted regressor. |
| *load* | Loads component at file path. |
| *name* | Returns string name of this component. |
| *needs_fitting* | Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances. |
| *parameters* | Returns the parameters which were used to initialize the component. |
| *predict* | Make predictions using selected features. |
| *predict_proba* | Make probability estimates for labels. |
| *save* | Saves component at file path. |
| *supported_problem_types* | Problem types this estimator supports. |
| *update_parameters* | Updates the parameter dictionary of the component. |

**clone**(*self*)

Constructs a new component with the same parameters and random state.

**Returns** A new instance of this component with identical parameters and random state.

**default_parameters**(*cls*)

Returns the default parameters for stacked ensemble classes.

**Returns** default parameters for this component.

**Return type** dict

**describe**(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

**Parameters**

- **print_name** (*bool, optional*) – whether to print name of component

- **return_dict** (*bool, optional*) – whether to return description as dictionary in the format {"name": name, "parameters": parameters}

**Returns** Returns dictionary if return_dict is True, else None.

**Return type** None or dict

**property feature_importance**(*self*)

Not implemented for StackedEnsembleClassifier and StackedEnsembleRegressor.

**fit**(*self, X: pandas.DataFrame, y: Optional[pandas.Series] = None*)

Fits estimator to data.

**Parameters**

- **X** (*pd.DataFrame*) – The input training data of shape [n_samples, n_features].
- **y** (*pd.Series, optional*) – The target training data of length [n_samples].

**Returns** self

**get_prediction_intervals**(*self, X: pandas.DataFrame, y: Optional[pandas.Series] = None, coverage: List[float] = None, predictions: pandas.Series = None*) → Dict[str, pandas.Series]

Find the prediction intervals using the fitted regressor.

This function takes the predictions of the fitted estimator and calculates the rolling standard deviation across all predictions using a window size of 5. The lower and upper predictions are determined by taking the percent point (quantile) function of the lower tail probability at each bound multiplied by the rolling standard deviation.

**Parameters**

- **X** (*pd.DataFrame*) – Data of shape [n_samples, n_features].
- **y** (*pd.Series*) – Target data. Ignored.
- **coverage** (*list[float]*) – A list of floats between the values 0 and 1 that the upper and lower bounds of the prediction interval should be calculated for.
- **predictions** (*pd.Series*) – Optional list of predictions to use. If None, will generate predictions using *X*.

**Returns** Prediction intervals, keys are in the format {coverage}_lower or {coverage}_upper.

**Return type** dict

**Raises** **MethodPropertyNotFoundError** – If the estimator does not support Time Series Regression as a problem type.

**static load**(*file_path*)

Loads component at file path.

**Parameters** **file_path** (*str*) – Location to load file.

**Returns** ComponentBase object

**property name**(*cls*)

Returns string name of this component.

**needs_fitting**(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need to be fit or whose fit methods do nothing.

**Returns** True.

**property parameters**(*self*)

Returns the parameters which were used to initialize the component.

**predict**(*self*, *X: pandas.DataFrame*) → pandas.Series

Make predictions using selected features.

**Parameters** **X** (*pd.DataFrame*) – Data of shape [n_samples, n_features].

**Returns** Predicted values.

**Return type** pd.Series

**Raises** **MethodPropertyNotFoundError** – If estimator does not have a predict method or a component_obj that implements predict.

**predict_proba**(*self*, *X: pandas.DataFrame*) → pandas.Series

Make probability estimates for labels.

**Parameters** **X** (*pd.DataFrame*) – Features.

**Returns** Probability estimates.

**Return type** pd.Series

**Raises** **MethodPropertyNotFoundError** – If estimator does not have a predict_proba method or a component_obj that implements predict_proba.

**save**(*self*, *file_path*, *pickle_protocol=cloudpickle.DEFAULT_PROTOCOL*)

Saves component at file path.

**Parameters**

- **file_path** (*str*) – Location to save file.
- **pickle_protocol** (*int*) – The pickle data stream format.

**property supported_problem_types**(*cls*)

Problem types this estimator supports.

**update_parameters**(*self*, *update_dict*, *reset_fit=True*)

Updates the parameter dictionary of the component.

**Parameters**

- **update_dict** (*dict*) – A dict of parameters to update.
- **reset_fit** (*bool, optional*) – If True, will set *_is_fitted* to False.

### stacked_ensemble_classifier

Stacked Ensemble Classifier.

### Module Contents

### Classes Summary

| | |
|---|---|
| *StackedEnsembleClassifier* | Stacked Ensemble Classifier. |

### Contents

**class** evalml.pipelines.components.ensemble.stacked_ensemble_classifier.**StackedEnsembleClassifier**(*final_est n_jobs= 1, random_see **kwarg*

Stacked Ensemble Classifier.

**Parameters**

- **final_estimator** (*Estimator or subclass*) – The classifier used to combine the base estimators. If None, uses ElasticNetClassifier.
- **n_jobs** (*int or None*) – Integer describing level of parallelism used for pipelines. None and 1 are equivalent. If set to -1, all CPUs are used. For n_jobs below -1, (n_cpus + 1 + n_jobs) are used. Defaults to -1. - Note: there could be some multi-process errors thrown for values of *n_jobs != 1*. If this is the case, please use *n_jobs = 1*.
- **random_seed** (*int*) – Seed for the random number generator. Defaults to 0.

### Example

```
>>> from evalml.pipelines.component_graph import ComponentGraph
>>> from evalml.pipelines.components.estimators.classifiers.decision_tree_
↪classifier import DecisionTreeClassifier
>>> from evalml.pipelines.components.estimators.classifiers.elasticnet_classifier␣
↪import ElasticNetClassifier
...
>>> component_graph = {
...     "Decision Tree": [DecisionTreeClassifier(random_seed=3), "X", "y"],
...     "Decision Tree B": [DecisionTreeClassifier(random_seed=4), "X", "y"],
...     "Stacked Ensemble": [
...         StackedEnsembleClassifier(n_jobs=1, final_
↪estimator=DecisionTreeClassifier()),
...         "Decision Tree.x",
...         "Decision Tree B.x",
...         "y",
...     ],
... }
...
>>> cg = ComponentGraph(component_graph)
>>> assert cg.default_parameters == {
```

(continues on next page)

(continued from previous page)

```
...     'Decision Tree Classifier': {'criterion': 'gini',
...                                  'max_features': 'auto',
...                                  'max_depth': 6,
...                                  'min_samples_split': 2,
...                                  'min_weight_fraction_leaf': 0.0},
...     'Stacked Ensemble Classifier': {'final_estimator': ElasticNetClassifier,
...                                     'n_jobs': -1}}
```

**Attributes**

| | |
|---|---|
| **hyper-parameter_ranges** | {} |
| **model_family** | ModelFamily.ENSEMBLE |
| **modifies_features** | True |
| **modifies_target** | False |
| **name** | Stacked Ensemble Classifier |
| **supported_problem_types** | [ ProblemTypes.BINARY, ProblemTypes.MULTICLASS, ProblemTypes.TIME_SERIES_BINARY, ProblemTypes.TIME_SERIES_MULTICLASS,] |
| **training_only** | False |

**Methods**

| | |
|---|---|
| *clone* | Constructs a new component with the same parameters and random state. |
| *default_parameters* | Returns the default parameters for stacked ensemble classes. |
| *describe* | Describe a component and its parameters. |
| *feature_importance* | Not implemented for StackedEnsembleClassifier and StackedEnsembleRegressor. |
| *fit* | Fits estimator to data. |
| *get_prediction_intervals* | Find the prediction intervals using the fitted regressor. |
| *load* | Loads component at file path. |
| *needs_fitting* | Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances. |
| *parameters* | Returns the parameters which were used to initialize the component. |
| *predict* | Make predictions using selected features. |
| *predict_proba* | Make probability estimates for labels. |
| *save* | Saves component at file path. |
| *update_parameters* | Updates the parameter dictionary of the component. |

**clone**(*self*)

Constructs a new component with the same parameters and random state.

**Returns** A new instance of this component with identical parameters and random state.

**default_parameters**(*cls*)

Returns the default parameters for stacked ensemble classes.

**Returns** default parameters for this component.

**Return type** dict

**describe**(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

**Parameters**

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description as dictionary in the format {"name": name, "parameters": parameters}

**Returns** Returns dictionary if return_dict is True, else None.

**Return type** None or dict

**property feature_importance**(*self*)

Not implemented for StackedEnsembleClassifier and StackedEnsembleRegressor.

**fit**(*self*, *X: pandas.DataFrame*, *y: Optional[pandas.Series] = None*)

Fits estimator to data.

**Parameters**

- **X** (*pd.DataFrame*) – The input training data of shape [n_samples, n_features].
- **y** (*pd.Series, optional*) – The target training data of length [n_samples].

**Returns** self

**get_prediction_intervals**(*self*, *X: pandas.DataFrame*, *y: Optional[pandas.Series] = None*, *coverage: List[float] = None*, *predictions: pandas.Series = None*) → Dict[str, pandas.Series]

Find the prediction intervals using the fitted regressor.

This function takes the predictions of the fitted estimator and calculates the rolling standard deviation across all predictions using a window size of 5. The lower and upper predictions are determined by taking the percent point (quantile) function of the lower tail probability at each bound multiplied by the rolling standard deviation.

**Parameters**

- **X** (*pd.DataFrame*) – Data of shape [n_samples, n_features].
- **y** (*pd.Series*) – Target data. Ignored.
- **coverage** (*list[float]*) – A list of floats between the values 0 and 1 that the upper and lower bounds of the prediction interval should be calculated for.
- **predictions** (*pd.Series*) – Optional list of predictions to use. If None, will generate predictions using *X*.

**Returns** Prediction intervals, keys are in the format {coverage}_lower or {coverage}_upper.

**Return type** dict

**Raises** **MethodPropertyNotFoundError** – If the estimator does not support Time Series Regression as a problem type.

**static load**(*file_path*)

Loads component at file path.

**Parameters** **file_path** (*str*) – Location to load file.

**Returns** ComponentBase object

**needs_fitting**(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need to be fit or whose fit methods do nothing.

**Returns** True.

**property parameters**(*self*)

Returns the parameters which were used to initialize the component.

**predict**(*self*, *X: pandas.DataFrame*) → pandas.Series

Make predictions using selected features.

**Parameters** **X** (*pd.DataFrame*) – Data of shape [n_samples, n_features].

**Returns** Predicted values.

**Return type** pd.Series

**Raises** **MethodPropertyNotFoundError** – If estimator does not have a predict method or a component_obj that implements predict.

**predict_proba**(*self*, *X: pandas.DataFrame*) → pandas.Series

Make probability estimates for labels.

**Parameters** **X** (*pd.DataFrame*) – Features.

**Returns** Probability estimates.

**Return type** pd.Series

**Raises** **MethodPropertyNotFoundError** – If estimator does not have a predict_proba method or a component_obj that implements predict_proba.

**save**(*self*, *file_path*, *pickle_protocol=cloudpickle.DEFAULT_PROTOCOL*)

Saves component at file path.

**Parameters**

- **file_path** (*str*) – Location to save file.
- **pickle_protocol** (*int*) – The pickle data stream format.

**update_parameters**(*self*, *update_dict*, *reset_fit=True*)

Updates the parameter dictionary of the component.

**Parameters**

- **update_dict** (*dict*) – A dict of parameters to update.
- **reset_fit** (*bool, optional*) – If True, will set *_is_fitted* to False.

### stacked_ensemble_regressor

Stacked Ensemble Regressor.

### Module Contents

### Classes Summary

| | |
|---|---|
| *StackedEnsembleRegressor* | Stacked Ensemble Regressor. |

### Contents

**class** evalml.pipelines.components.ensemble.stacked_ensemble_regressor.**StackedEnsembleRegressor**(*final_estim n_jobs=-1, random_seed= **kwargs*)

Stacked Ensemble Regressor.

**Parameters**

- **final_estimator** (*Estimator or subclass*) – The regressor used to combine the base estimators. If None, uses ElasticNetRegressor.
- **n_jobs** (*int or None*) – Integer describing level of parallelism used for pipelines. None and 1 are equivalent. If set to -1, all CPUs are used. For n_jobs greater than -1, (n_cpus + 1 + n_jobs) are used. Defaults to -1. - Note: there could be some multi-process errors thrown for values of *n_jobs != 1*. If this is the case, please use *n_jobs = 1*.
- **random_seed** (*int*) – Seed for the random number generator. Defaults to 0.

### Example

```
>>> from evalml.pipelines.component_graph import ComponentGraph
>>> from evalml.pipelines.components.estimators.regressors.rf_regressor import␣
↪RandomForestRegressor
>>> from evalml.pipelines.components.estimators.regressors.elasticnet_regressor␣
↪import ElasticNetRegressor
...
>>> component_graph = {
...     "Random Forest": [RandomForestRegressor(random_seed=3), "X", "y"],
...     "Random Forest B": [RandomForestRegressor(random_seed=4), "X", "y"],
...     "Stacked Ensemble": [
...         StackedEnsembleRegressor(n_jobs=1, final_
↪estimator=RandomForestRegressor()),
...         "Random Forest.x",
...         "Random Forest B.x",
...         "y",
```

(continues on next page)

(continued from previous page)

```
...     ],
... }
...
>>> cg = ComponentGraph(component_graph)
>>> assert cg.default_parameters == {
...     'Random Forest Regressor': {'n_estimators': 100,
...                                 'max_depth': 6,
...                                 'n_jobs': -1},
...     'Stacked Ensemble Regressor': {'final_estimator': ElasticNetRegressor,
...                                    'n_jobs': -1}}
```

**Attributes**

| | |
|---|---|
| **hyper-parameter_ranges** | {} |
| **model_family** | ModelFamily.ENSEMBLE |
| **modifies_features** | True |
| **modifies_target** | False |
| **name** | Stacked Ensemble Regressor |
| **supported_problem_types** | [ ProblemTypes.REGRESSION, ProblemTypes.TIME_SERIES_REGRESSION,] |
| **training_only** | False |

**Methods**

| | |
|---|---|
| *clone* | Constructs a new component with the same parameters and random state. |
| *default_parameters* | Returns the default parameters for stacked ensemble classes. |
| *describe* | Describe a component and its parameters. |
| *feature_importance* | Not implemented for StackedEnsembleClassifier and StackedEnsembleRegressor. |
| *fit* | Fits estimator to data. |
| *get_prediction_intervals* | Find the prediction intervals using the fitted regressor. |
| *load* | Loads component at file path. |
| *needs_fitting* | Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances. |
| *parameters* | Returns the parameters which were used to initialize the component. |
| *predict* | Make predictions using selected features. |
| *predict_proba* | Make probability estimates for labels. |
| *save* | Saves component at file path. |
| *update_parameters* | Updates the parameter dictionary of the component. |

**clone**(*self*)

Constructs a new component with the same parameters and random state.

**Returns** A new instance of this component with identical parameters and random state.

**default_parameters**(*cls*)

Returns the default parameters for stacked ensemble classes.

**Returns** default parameters for this component.

**Return type** dict

**describe**(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

**Parameters**

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description as dictionary in the format {"name": name, "parameters": parameters}

**Returns** Returns dictionary if return_dict is True, else None.

**Return type** None or dict

**property feature_importance**(*self*)

Not implemented for StackedEnsembleClassifier and StackedEnsembleRegressor.

**fit**(*self*, *X: pandas.DataFrame*, *y: Optional[pandas.Series] = None*)

Fits estimator to data.

**Parameters**

- **X** (*pd.DataFrame*) – The input training data of shape [n_samples, n_features].
- **y** (*pd.Series, optional*) – The target training data of length [n_samples].

**Returns** self

**get_prediction_intervals**(*self*, *X: pandas.DataFrame*, *y: Optional[pandas.Series] = None*, *coverage: List[float] = None*, *predictions: pandas.Series = None*) → Dict[str, pandas.Series]

Find the prediction intervals using the fitted regressor.

This function takes the predictions of the fitted estimator and calculates the rolling standard deviation across all predictions using a window size of 5. The lower and upper predictions are determined by taking the percent point (quantile) function of the lower tail probability at each bound multiplied by the rolling standard deviation.

**Parameters**

- **X** (*pd.DataFrame*) – Data of shape [n_samples, n_features].
- **y** (*pd.Series*) – Target data. Ignored.
- **coverage** (*list[float]*) – A list of floats between the values 0 and 1 that the upper and lower bounds of the prediction interval should be calculated for.
- **predictions** (*pd.Series*) – Optional list of predictions to use. If None, will generate predictions using *X*.

**Returns** Prediction intervals, keys are in the format {coverage}_lower or {coverage}_upper.

**Return type** dict

**Raises** `MethodPropertyNotFoundError` – If the estimator does not support Time Series Regression as a problem type.

`static load`(*file_path*)

Loads component at file path.

**Parameters** `file_path` (*str*) – Location to load file.

**Returns** ComponentBase object

`needs_fitting`(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need to be fit or whose fit methods do nothing.

**Returns** True.

`property parameters`(*self*)

Returns the parameters which were used to initialize the component.

`predict`(*self*, *X: pandas.DataFrame*) → pandas.Series

Make predictions using selected features.

**Parameters** `X` (*pd.DataFrame*) – Data of shape [n_samples, n_features].

**Returns** Predicted values.

**Return type** pd.Series

**Raises** `MethodPropertyNotFoundError` – If estimator does not have a predict method or a component_obj that implements predict.

`predict_proba`(*self*, *X: pandas.DataFrame*) → pandas.Series

Make probability estimates for labels.

**Parameters** `X` (*pd.DataFrame*) – Features.

**Returns** Probability estimates.

**Return type** pd.Series

**Raises** `MethodPropertyNotFoundError` – If estimator does not have a predict_proba method or a component_obj that implements predict_proba.

`save`(*self*, *file_path*, *pickle_protocol=cloudpickle.DEFAULT_PROTOCOL*)

Saves component at file path.

**Parameters**

- `file_path` (*str*) – Location to save file.
- `pickle_protocol` (*int*) – The pickle data stream format.

`update_parameters`(*self*, *update_dict*, *reset_fit=True*)

Updates the parameter dictionary of the component.

**Parameters**

- `update_dict` (*dict*) – A dict of parameters to update.
- `reset_fit` (*bool, optional*) – If True, will set *_is_fitted* to False.

### Package Contents

### Classes Summary

| | |
|---|---|
| *StackedEnsembleBase* | Stacked Ensemble Base Class. |
| *StackedEnsembleClassifier* | Stacked Ensemble Classifier. |
| *StackedEnsembleRegressor* | Stacked Ensemble Regressor. |

### Contents

**class** evalml.pipelines.components.ensemble.**StackedEnsembleBase**(*final_estimator=None*, *n_jobs=-1*, *random_seed=0*, ***kwargs*)

Stacked Ensemble Base Class.

**Parameters**

- **final_estimator** (*Estimator or subclass*) – The estimator used to combine the base estimators.
- **n_jobs** (*int or None*) – Integer describing level of parallelism used for pipelines. None and 1 are equivalent. If set to -1, all CPUs are used. For n_jobs greater than -1, (n_cpus + 1 + n_jobs) are used. Defaults to -1. - Note: there could be some multi-process errors thrown for values of *n_jobs != 1*. If this is the case, please use *n_jobs = 1*.
- **random_seed** (*int*) – Seed for the random number generator. Defaults to 0.

**Attributes**

| | |
|---|---|
| **model_family** | ModelFamily.ENSEMBLE |
| **modifies_features** | True |
| **modifies_target** | False |
| **training_only** | False |

**Methods**

| | |
|---|---|
| *clone* | Constructs a new component with the same parameters and random state. |
| *default_parameters* | Returns the default parameters for stacked ensemble classes. |
| *describe* | Describe a component and its parameters. |
| *feature_importance* | Not implemented for StackedEnsembleClassifier and StackedEnsembleRegressor. |
| *fit* | Fits estimator to data. |
| *get_prediction_intervals* | Find the prediction intervals using the fitted regressor. |
| *load* | Loads component at file path. |
| *name* | Returns string name of this component. |
| *needs_fitting* | Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances. |
| *parameters* | Returns the parameters which were used to initialize the component. |
| *predict* | Make predictions using selected features. |
| *predict_proba* | Make probability estimates for labels. |
| *save* | Saves component at file path. |
| *supported_problem_types* | Problem types this estimator supports. |
| *update_parameters* | Updates the parameter dictionary of the component. |

**clone**(*self*)

Constructs a new component with the same parameters and random state.

**Returns** A new instance of this component with identical parameters and random state.

**default_parameters**(*cls*)

Returns the default parameters for stacked ensemble classes.

**Returns** default parameters for this component.

**Return type** dict

**describe**(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

**Parameters**

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description as dictionary in the format {"name": name, "parameters": parameters}

**Returns** Returns dictionary if return_dict is True, else None.

**Return type** None or dict

**property feature_importance**(*self*)

Not implemented for StackedEnsembleClassifier and StackedEnsembleRegressor.

**fit**(*self*, *X: pandas.DataFrame*, *y: Optional[pandas.Series] = None*)

Fits estimator to data.

**Parameters**

- **X** (*pd.DataFrame*) – The input training data of shape [n_samples, n_features].

- **y** (*pd.Series, optional*) – The target training data of length [n_samples].

**Returns** self

**get_prediction_intervals**(*self, X: pandas.DataFrame, y: Optional[pandas.Series] = None, coverage: List[float] = None, predictions: pandas.Series = None*) → Dict[str, pandas.Series]

Find the prediction intervals using the fitted regressor.

This function takes the predictions of the fitted estimator and calculates the rolling standard deviation across all predictions using a window size of 5. The lower and upper predictions are determined by taking the percent point (quantile) function of the lower tail probability at each bound multiplied by the rolling standard deviation.

**Parameters**

- **X** (*pd.DataFrame*) – Data of shape [n_samples, n_features].
- **y** (*pd.Series*) – Target data. Ignored.
- **coverage** (*list[float]*) – A list of floats between the values 0 and 1 that the upper and lower bounds of the prediction interval should be calculated for.
- **predictions** (*pd.Series*) – Optional list of predictions to use. If None, will generate predictions using *X*.

**Returns** Prediction intervals, keys are in the format {coverage}_lower or {coverage}_upper.

**Return type** dict

**Raises** **MethodPropertyNotFoundError** – If the estimator does not support Time Series Regression as a problem type.

**static load**(*file_path*)

Loads component at file path.

**Parameters** **file_path** (*str*) – Location to load file.

**Returns** ComponentBase object

**property name**(*cls*)

Returns string name of this component.

**needs_fitting**(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need to be fit or whose fit methods do nothing.

**Returns** True.

**property parameters**(*self*)

Returns the parameters which were used to initialize the component.

**predict**(*self, X: pandas.DataFrame*) → pandas.Series

Make predictions using selected features.

**Parameters** **X** (*pd.DataFrame*) – Data of shape [n_samples, n_features].

**Returns** Predicted values.

**Return type** pd.Series

**Raises** **MethodPropertyNotFoundError** – If estimator does not have a predict method or a component_obj that implements predict.

**predict_proba**(*self*, *X: pandas.DataFrame*) → pandas.Series

Make probability estimates for labels.

**Parameters** **X** (*pd.DataFrame*) – Features.

**Returns** Probability estimates.

**Return type** pd.Series

**Raises** **MethodPropertyNotFoundError** – If estimator does not have a predict_proba method or a component_obj that implements predict_proba.

**save**(*self*, *file_path*, *pickle_protocol=cloudpickle.DEFAULT_PROTOCOL*)

Saves component at file path.

**Parameters**

- **file_path** (*str*) – Location to save file.
- **pickle_protocol** (*int*) – The pickle data stream format.

**property supported_problem_types**(*cls*)

Problem types this estimator supports.

**update_parameters**(*self*, *update_dict*, *reset_fit=True*)

Updates the parameter dictionary of the component.

**Parameters**

- **update_dict** (*dict*) – A dict of parameters to update.
- **reset_fit** (*bool, optional*) – If True, will set *_is_fitted* to False.

**class** evalml.pipelines.components.ensemble.**StackedEnsembleClassifier**(*final_estimator=None*, *n_jobs=- 1*, *random_seed=0*, ***kwargs*)

Stacked Ensemble Classifier.

**Parameters**

- **final_estimator** (*Estimator or subclass*) – The classifier used to combine the base estimators. If None, uses ElasticNetClassifier.
- **n_jobs** (*int or None*) – Integer describing level of parallelism used for pipelines. None and 1 are equivalent. If set to -1, all CPUs are used. For n_jobs below -1, (n_cpus + 1 + n_jobs) are used. Defaults to -1. - Note: there could be some multi-process errors thrown for values of *n_jobs != 1*. If this is the case, please use *n_jobs = 1*.
- **random_seed** (*int*) – Seed for the random number generator. Defaults to 0.

### Example

```
>>> from evalml.pipelines.component_graph import ComponentGraph
>>> from evalml.pipelines.components.estimators.classifiers.decision_tree_
↪classifier import DecisionTreeClassifier
>>> from evalml.pipelines.components.estimators.classifiers.elasticnet_classifier␣
↪import ElasticNetClassifier
...
>>> component_graph = {
```

(continues on next page)

(continued from previous page)

```
...     "Decision Tree": [DecisionTreeClassifier(random_seed=3), "X", "y"],
...     "Decision Tree B": [DecisionTreeClassifier(random_seed=4), "X", "y"],
...     "Stacked Ensemble": [
...         StackedEnsembleClassifier(n_jobs=1, final_
↪estimator=DecisionTreeClassifier()),
...         "Decision Tree.x",
...         "Decision Tree B.x",
...         "y",
...     ],
... }
...
>>> cg = ComponentGraph(component_graph)
>>> assert cg.default_parameters == {
...     'Decision Tree Classifier': {'criterion': 'gini',
...                                  'max_features': 'auto',
...                                  'max_depth': 6,
...                                  'min_samples_split': 2,
...                                  'min_weight_fraction_leaf': 0.0},
...     'Stacked Ensemble Classifier': {'final_estimator': ElasticNetClassifier,
...                                     'n_jobs': -1}}
```

**Attributes**

| | |
|---|---|
| **hyper-parameter_ranges** | {} |
| **model_family** | ModelFamily.ENSEMBLE |
| **modifies_features** | True |
| **modifies_target** | False |
| **name** | Stacked Ensemble Classifier |
| **supported_problem_types** | [ ProblemTypes.BINARY, ProblemTypes.MULTICLASS, ProblemTypes.TIME_SERIES_BINARY, ProblemTypes.TIME_SERIES_MULTICLASS,] |
| **training_only** | False |

**Methods**

| | |
|---|---|
| *clone* | Constructs a new component with the same parameters and random state. |
| *default_parameters* | Returns the default parameters for stacked ensemble classes. |
| *describe* | Describe a component and its parameters. |
| *feature_importance* | Not implemented for StackedEnsembleClassifier and StackedEnsembleRegressor. |
| *fit* | Fits estimator to data. |
| *get_prediction_intervals* | Find the prediction intervals using the fitted regressor. |
| *load* | Loads component at file path. |
| *needs_fitting* | Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances. |
| *parameters* | Returns the parameters which were used to initialize the component. |
| *predict* | Make predictions using selected features. |
| *predict_proba* | Make probability estimates for labels. |
| *save* | Saves component at file path. |
| *update_parameters* | Updates the parameter dictionary of the component. |

**clone**(*self*)

Constructs a new component with the same parameters and random state.

**Returns** A new instance of this component with identical parameters and random state.

**default_parameters**(*cls*)

Returns the default parameters for stacked ensemble classes.

**Returns** default parameters for this component.

**Return type** dict

**describe**(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

**Parameters**

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description as dictionary in the format {"name": name, "parameters": parameters}

**Returns** Returns dictionary if return_dict is True, else None.

**Return type** None or dict

**property feature_importance**(*self*)

Not implemented for StackedEnsembleClassifier and StackedEnsembleRegressor.

**fit**(*self*, *X: pandas.DataFrame*, *y: Optional[pandas.Series] = None*)

Fits estimator to data.

**Parameters**

- **X** (*pd.DataFrame*) – The input training data of shape [n_samples, n_features].
- **y** (*pd.Series, optional*) – The target training data of length [n_samples].

**Returns** self

**get_prediction_intervals**(*self*, *X: pandas.DataFrame*, *y: Optional[pandas.Series] = None*, *coverage: List[float] = None*, *predictions: pandas.Series = None*) → Dict[str, pandas.Series]

Find the prediction intervals using the fitted regressor.

This function takes the predictions of the fitted estimator and calculates the rolling standard deviation across all predictions using a window size of 5. The lower and upper predictions are determined by taking the percent point (quantile) function of the lower tail probability at each bound multiplied by the rolling standard deviation.

**Parameters**

- **X** (*pd.DataFrame*) – Data of shape [n_samples, n_features].
- **y** (*pd.Series*) – Target data. Ignored.
- **coverage** (*list[float]*) – A list of floats between the values 0 and 1 that the upper and lower bounds of the prediction interval should be calculated for.
- **predictions** (*pd.Series*) – Optional list of predictions to use. If None, will generate predictions using *X*.

**Returns** Prediction intervals, keys are in the format {coverage}_lower or {coverage}_upper.

**Return type** dict

**Raises** **MethodPropertyNotFoundError** – If the estimator does not support Time Series Regression as a problem type.

**static load**(*file_path*)

Loads component at file path.

**Parameters** **file_path** (*str*) – Location to load file.

**Returns** ComponentBase object

**needs_fitting**(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need to be fit or whose fit methods do nothing.

**Returns** True.

**property parameters**(*self*)

Returns the parameters which were used to initialize the component.

**predict**(*self*, *X: pandas.DataFrame*) → pandas.Series

Make predictions using selected features.

**Parameters** **X** (*pd.DataFrame*) – Data of shape [n_samples, n_features].

**Returns** Predicted values.

**Return type** pd.Series

**Raises** **MethodPropertyNotFoundError** – If estimator does not have a predict method or a component_obj that implements predict.

**predict_proba**(*self*, *X: pandas.DataFrame*) → pandas.Series

Make probability estimates for labels.

**Parameters** **X** (*pd.DataFrame*) – Features.

**Returns** Probability estimates.

**Return type** pd.Series

**Raises** **MethodPropertyNotFoundError** – If estimator does not have a predict_proba method or a component_obj that implements predict_proba.

**save**(*self*, *file_path*, *pickle_protocol=cloudpickle.DEFAULT_PROTOCOL*)

Saves component at file path.

**Parameters**

- **file_path** (*str*) – Location to save file.
- **pickle_protocol** (*int*) – The pickle data stream format.

**update_parameters**(*self*, *update_dict*, *reset_fit=True*)

Updates the parameter dictionary of the component.

**Parameters**

- **update_dict** (*dict*) – A dict of parameters to update.
- **reset_fit** (*bool, optional*) – If True, will set *_is_fitted* to False.

**class** evalml.pipelines.components.ensemble.**StackedEnsembleRegressor**(*final_estimator=None*, *n_jobs=- 1*, *random_seed=0*, ***kwargs*)

Stacked Ensemble Regressor.

**Parameters**

- **final_estimator** (*Estimator or subclass*) – The regressor used to combine the base estimators. If None, uses ElasticNetRegressor.
- **n_jobs** (*int or None*) – Integer describing level of parallelism used for pipelines. None and 1 are equivalent. If set to -1, all CPUs are used. For n_jobs greater than -1, (n_cpus + 1 + n_jobs) are used. Defaults to -1. - Note: there could be some multi-process errors thrown for values of *n_jobs != 1*. If this is the case, please use *n_jobs = 1*.
- **random_seed** (*int*) – Seed for the random number generator. Defaults to 0.

### Example

```
>>> from evalml.pipelines.component_graph import ComponentGraph
>>> from evalml.pipelines.components.estimators.regressors.rf_regressor import␣
↪RandomForestRegressor
>>> from evalml.pipelines.components.estimators.regressors.elasticnet_regressor␣
↪import ElasticNetRegressor
...
>>> component_graph = {
...     "Random Forest": [RandomForestRegressor(random_seed=3), "X", "y"],
...     "Random Forest B": [RandomForestRegressor(random_seed=4), "X", "y"],
...     "Stacked Ensemble": [
```

(continues on next page)

(continued from previous page)

```
...         StackedEnsembleRegressor(n_jobs=1, final_
↪estimator=RandomForestRegressor()),
...         "Random Forest.x",
...         "Random Forest B.x",
...         "y",
...     ],
... }
...
>>> cg = ComponentGraph(component_graph)
>>> assert cg.default_parameters == {
...     'Random Forest Regressor': {'n_estimators': 100,
...                                 'max_depth': 6,
...                                 'n_jobs': -1},
...     'Stacked Ensemble Regressor': {'final_estimator': ElasticNetRegressor,
...                                    'n_jobs': -1}}
```

**Attributes**

| | |
|---|---|
| **hyper-parameter_ranges** | {} |
| **model_family** | ModelFamily.ENSEMBLE |
| **modifies_features** | True |
| **modifies_target** | False |
| **name** | Stacked Ensemble Regressor |
| **supported_problem_types** | [ ProblemTypes.REGRESSION, ProblemTypes.TIME_SERIES_REGRESSION,] |
| **training_only** | False |

**Methods**

| | |
|---|---|
| *clone* | Constructs a new component with the same parameters and random state. |
| *default_parameters* | Returns the default parameters for stacked ensemble classes. |
| *describe* | Describe a component and its parameters. |
| *feature_importance* | Not implemented for StackedEnsembleClassifier and StackedEnsembleRegressor. |
| *fit* | Fits estimator to data. |
| *get_prediction_intervals* | Find the prediction intervals using the fitted regressor. |
| *load* | Loads component at file path. |
| *needs_fitting* | Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances. |
| *parameters* | Returns the parameters which were used to initialize the component. |
| *predict* | Make predictions using selected features. |
| *predict_proba* | Make probability estimates for labels. |
| *save* | Saves component at file path. |
| *update_parameters* | Updates the parameter dictionary of the component. |

**clone**(*self*)

Constructs a new component with the same parameters and random state.

**Returns** A new instance of this component with identical parameters and random state.

**default_parameters**(*cls*)

Returns the default parameters for stacked ensemble classes.

**Returns** default parameters for this component.

**Return type** dict

**describe**(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

**Parameters**

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description as dictionary in the format {"name": name, "parameters": parameters}

**Returns** Returns dictionary if return_dict is True, else None.

**Return type** None or dict

**property feature_importance**(*self*)

Not implemented for StackedEnsembleClassifier and StackedEnsembleRegressor.

**fit**(*self*, *X: pandas.DataFrame*, *y: Optional[pandas.Series] = None*)

Fits estimator to data.

**Parameters**

- **X** (*pd.DataFrame*) – The input training data of shape [n_samples, n_features].
- **y** (*pd.Series, optional*) – The target training data of length [n_samples].

**Returns** self

**get_prediction_intervals**(*self*, *X: pandas.DataFrame*, *y: Optional[pandas.Series] = None*, *coverage: List[float] = None*, *predictions: pandas.Series = None*) → Dict[str, pandas.Series]

Find the prediction intervals using the fitted regressor.

This function takes the predictions of the fitted estimator and calculates the rolling standard deviation across all predictions using a window size of 5. The lower and upper predictions are determined by taking the percent point (quantile) function of the lower tail probability at each bound multiplied by the rolling standard deviation.

**Parameters**

- **X** (*pd.DataFrame*) – Data of shape [n_samples, n_features].
- **y** (*pd.Series*) – Target data. Ignored.
- **coverage** (*list[float]*) – A list of floats between the values 0 and 1 that the upper and lower bounds of the prediction interval should be calculated for.
- **predictions** (*pd.Series*) – Optional list of predictions to use. If None, will generate predictions using *X*.

**Returns** Prediction intervals, keys are in the format {coverage}_lower or {coverage}_upper.

**Return type** dict

**Raises** **MethodPropertyNotFoundError** – If the estimator does not support Time Series Regression as a problem type.

**static load**(*file_path*)

Loads component at file path.

**Parameters** **file_path** (*str*) – Location to load file.

**Returns** ComponentBase object

**needs_fitting**(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need to be fit or whose fit methods do nothing.

**Returns** True.

**property parameters**(*self*)

Returns the parameters which were used to initialize the component.

**predict**(*self*, *X: pandas.DataFrame*) → pandas.Series

Make predictions using selected features.

**Parameters** **X** (*pd.DataFrame*) – Data of shape [n_samples, n_features].

**Returns** Predicted values.

**Return type** pd.Series

**Raises** **MethodPropertyNotFoundError** – If estimator does not have a predict method or a component_obj that implements predict.

**predict_proba**(*self*, *X: pandas.DataFrame*) → pandas.Series

Make probability estimates for labels.

**Parameters** **X** (*pd.DataFrame*) – Features.

**Returns** Probability estimates.

**Return type** pd.Series

**Raises** **MethodPropertyNotFoundError** – If estimator does not have a predict_proba method or a component_obj that implements predict_proba.

**save**(*self*, *file_path*, *pickle_protocol=cloudpickle.DEFAULT_PROTOCOL*)

Saves component at file path.

**Parameters**

- **file_path** (*str*) – Location to save file.
- **pickle_protocol** (*int*) – The pickle data stream format.

**update_parameters**(*self*, *update_dict*, *reset_fit=True*)

Updates the parameter dictionary of the component.

**Parameters**

- **update_dict** (*dict*) – A dict of parameters to update.
- **reset_fit** (*bool, optional*) – If True, will set *_is_fitted* to False.

## estimators

EvalML estimator components.

## Subpackages

## classifiers

Classification model components.

## Submodules

## baseline_classifier

Baseline classifier.

### Module Contents

### Classes Summary

| | |
|---|---|
| *BaselineClassifier* | Classifier that predicts using the specified strategy. |

### Contents

**class** evalml.pipelines.components.estimators.classifiers.baseline_classifier.**BaselineClassifier**(*strategy=*
*random_seed*
***kwargs*

Classifier that predicts using the specified strategy.

This is useful as a simple baseline classifier to compare with other classifiers.

**Parameters**

- **strategy** (*str*) – Method used to predict. Valid options are "mode", "random" and "random_weighted". Defaults to "mode".
- **random_seed** (*int*) – Seed for the random number generator. Defaults to 0.

**Attributes**

| | |
|---|---|
| **hyperparameter_ranges** | {} |
| **model_family** | ModelFamily.BASELINE |
| **modifies_features** | True |
| **modifies_target** | False |
| **name** | Baseline Classifier |
| **supported_problem_types** | [ProblemTypes.BINARY, ProblemTypes.MULTICLASS] |
| **training_only** | False |

**Methods**

| | |
|---|---|
| *classes_* | Returns class labels. Will return None before fitting. |
| *clone* | Constructs a new component with the same parameters and random state. |
| *default_parameters* | Returns the default parameters for this component. |
| *describe* | Describe a component and its parameters. |
| *feature_importance* | Returns importance associated with each feature. Since baseline classifiers do not use input features to calculate predictions, returns an array of zeroes. |
| *fit* | Fits baseline classifier component to data. |
| *get_prediction_intervals* | Find the prediction intervals using the fitted regressor. |
| *load* | Loads component at file path. |
| *needs_fitting* | Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances. |
| *parameters* | Returns the parameters which were used to initialize the component. |
| *predict* | Make predictions using the baseline classification strategy. |
| *predict_proba* | Make prediction probabilities using the baseline classification strategy. |
| *save* | Saves component at file path. |
| *update_parameters* | Updates the parameter dictionary of the component. |

**property classes_**(*self*)

Returns class labels. Will return None before fitting.

**Returns** Class names

**Return type** list[str] or list(float)

**clone**(*self*)

Constructs a new component with the same parameters and random state.

**Returns** A new instance of this component with identical parameters and random state.

**default_parameters**(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

**Returns** Default parameters for this component.

**Return type** dict

**describe**(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

**Parameters**

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description as dictionary in the format {"name": name, "parameters": parameters}

**Returns** Returns dictionary if return_dict is True, else None.

**Return type** None or dict

**property feature_importance**(*self*)

Returns importance associated with each feature. Since baseline classifiers do not use input features to calculate predictions, returns an array of zeroes.

**Returns** An array of zeroes

**Return type** pd.Series

**fit**(*self*, *X*, *y=None*)

Fits baseline classifier component to data.

**Parameters**

- **X** (*pd.DataFrame*) – The input training data of shape [n_samples, n_features].
- **y** (*pd.Series*) – The target training data of length [n_samples].

**Returns** self

**Raises** **ValueError** – If y is None.

**get_prediction_intervals**(*self*, *X: pandas.DataFrame*, *y: Optional[pandas.Series] = None*, *coverage: List[float] = None*, *predictions: pandas.Series = None*) → Dict[str, pandas.Series]

Find the prediction intervals using the fitted regressor.

This function takes the predictions of the fitted estimator and calculates the rolling standard deviation across all predictions using a window size of 5. The lower and upper predictions are determined by taking the percent point (quantile) function of the lower tail probability at each bound multiplied by the rolling standard deviation.

**Parameters**

- **X** (*pd.DataFrame*) – Data of shape [n_samples, n_features].
- **y** (*pd.Series*) – Target data. Ignored.
- **coverage** (*list[float]*) – A list of floats between the values 0 and 1 that the upper and lower bounds of the prediction interval should be calculated for.
- **predictions** (*pd.Series*) – Optional list of predictions to use. If None, will generate predictions using *X*.

**Returns** Prediction intervals, keys are in the format {coverage}_lower or {coverage}_upper.

**Return type** dict

**Raises** **MethodPropertyNotFoundError** – If the estimator does not support Time Series Regression as a problem type.

**static load**(*file_path*)

Loads component at file path.

**Parameters** **file_path** (*str*) – Location to load file.

**Returns** ComponentBase object

**needs_fitting**(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need to be fit or whose fit methods do nothing.

**Returns** True.

**property parameters**(*self*)

Returns the parameters which were used to initialize the component.

**predict**(*self*, *X*)

Make predictions using the baseline classification strategy.

**Parameters** **X** (*pd.DataFrame*) – Data of shape [n_samples, n_features].

**Returns** Predicted values.

**Return type** pd.Series

**predict_proba**(*self*, *X*)

Make prediction probabilities using the baseline classification strategy.

**Parameters** **X** (*pd.DataFrame*) – Data of shape [n_samples, n_features].

**Returns** Predicted probability values.

**Return type** pd.DataFrame

**save**(*self*, *file_path*, *pickle_protocol=cloudpickle.DEFAULT_PROTOCOL*)

Saves component at file path.

**Parameters**

- **file_path** (*str*) – Location to save file.
- **pickle_protocol** (*int*) – The pickle data stream format.

**update_parameters**(*self*, *update_dict*, *reset_fit=True*)

Updates the parameter dictionary of the component.

**Parameters**

- **update_dict** (*dict*) – A dict of parameters to update.
- **reset_fit** (*bool, optional*) – If True, will set *_is_fitted* to False.

## catboost_classifier

CatBoost Classifier, a classifier that uses gradient-boosting on decision trees. CatBoost is an open-source library and natively supports categorical features.

### Module Contents

### Classes Summary

| | |
|---|---|
| *CatBoostClassifier* | CatBoost Classifier, a classifier that uses gradient-boosting on decision trees. CatBoost is an open-source library and natively supports categorical features. |

### Contents

**class** evalml.pipelines.components.estimators.classifiers.catboost_classifier.**CatBoostClassifier**(*n_estimat eta=0.03, max_dept boot- strap_typ silent=Tr al- low_writi ran- dom_seed n_jobs=- 1, **kwargs*

CatBoost Classifier, a classifier that uses gradient-boosting on decision trees. CatBoost is an open-source library and natively supports categorical features.

For more information, check out https://catboost.ai/

**Parameters**

- **n_estimators** (*float*) – The maximum number of trees to build. Defaults to 10.
- **eta** (*float*) – The learning rate. Defaults to 0.03.
- **max_depth** (*int*) – The maximum tree depth for base learners. Defaults to 6.
- **bootstrap_type** (*string*) – Defines the method for sampling the weights of objects. Available methods are 'Bayesian', 'Bernoulli', 'MVS'. Defaults to None.
- **silent** (*boolean*) – Whether to use the "silent" logging mode. Defaults to True.
- **allow_writing_files** (*boolean*) – Whether to allow writing snapshot files while training. Defaults to False.
- **n_jobs** (*int or None*) – Number of jobs to run in parallel. -1 uses all processes. Defaults to -1.
- **random_seed** (*int*) – Seed for the random number generator. Defaults to 0.

**Attributes**

| | |
|---|---|
| **hyper-parameter_ranges** | { "n_estimators": Integer(4, 100), "eta": Real(0.000001, 1), "max_depth": Integer(4, 10),} |
| **model_family** | ModelFamily.CATBOOST |
| **modifies_features** | True |
| **modifies_target** | False |
| **name** | CatBoost Classifier |
| **supported_problem_types** | [ ProblemTypes.BINARY, ProblemTypes.MULTICLASS, ProblemTypes.TIME_SERIES_BINARY, ProblemTypes.TIME_SERIES_MULTICLASS,] |
| **training_only** | False |

**Methods**

| | |
|---|---|
| *clone* | Constructs a new component with the same parameters and random state. |
| *default_parameters* | Returns the default parameters for this component. |
| *describe* | Describe a component and its parameters. |
| *feature_importance* | Feature importance of fitted CatBoost classifier. |
| *fit* | Fits CatBoost classifier component to data. |
| *get_prediction_intervals* | Find the prediction intervals using the fitted regressor. |
| *load* | Loads component at file path. |
| *needs_fitting* | Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances. |
| *parameters* | Returns the parameters which were used to initialize the component. |
| *predict* | Make predictions using the fitted CatBoost classifier. |
| *predict_proba* | Make prediction probabilities using the fitted CatBoost classifier. |
| *save* | Saves component at file path. |
| *update_parameters* | Updates the parameter dictionary of the component. |

**clone**(*self*)

Constructs a new component with the same parameters and random state.

**Returns** A new instance of this component with identical parameters and random state.

**default_parameters**(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

**Returns** Default parameters for this component.

**Return type** dict

**describe**(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

**Parameters**

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description as dictionary in the format {"name": name, "parameters": parameters}

**Returns** Returns dictionary if return_dict is True, else None.

**Return type** None or dict

**property feature_importance**(*self*)

Feature importance of fitted CatBoost classifier.

**fit**(*self*, *X*, *y=None*)

Fits CatBoost classifier component to data.

**Parameters**

- **X** (*pd.DataFrame*) – The input training data of shape [n_samples, n_features].

- **y** (*pd.Series*) – The target training data of length [n_samples].

**Returns** self

**get_prediction_intervals**(*self, X: pandas.DataFrame, y: Optional[pandas.Series] = None, coverage: List[float] = None, predictions: pandas.Series = None*) → Dict[str, pandas.Series]

Find the prediction intervals using the fitted regressor.

This function takes the predictions of the fitted estimator and calculates the rolling standard deviation across all predictions using a window size of 5. The lower and upper predictions are determined by taking the percent point (quantile) function of the lower tail probability at each bound multiplied by the rolling standard deviation.

**Parameters**

- **X** (*pd.DataFrame*) – Data of shape [n_samples, n_features].
- **y** (*pd.Series*) – Target data. Ignored.
- **coverage** (*list[float]*) – A list of floats between the values 0 and 1 that the upper and lower bounds of the prediction interval should be calculated for.
- **predictions** (*pd.Series*) – Optional list of predictions to use. If None, will generate predictions using *X*.

**Returns** Prediction intervals, keys are in the format {coverage}_lower or {coverage}_upper.

**Return type** dict

**Raises** **MethodPropertyNotFoundError** – If the estimator does not support Time Series Regression as a problem type.

**static load**(*file_path*)

Loads component at file path.

**Parameters** **file_path** (*str*) – Location to load file.

**Returns** ComponentBase object

**needs_fitting**(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need to be fit or whose fit methods do nothing.

**Returns** True.

**property parameters**(*self*)

Returns the parameters which were used to initialize the component.

**predict**(*self, X*)

Make predictions using the fitted CatBoost classifier.

**Parameters** **X** (*pd.DataFrame*) – Data of shape [n_samples, n_features].

**Returns** Predicted values.

**Return type** pd.Series

**predict_proba**(*self, X*)

Make prediction probabilities using the fitted CatBoost classifier.

**Parameters** **X** (*pd.DataFrame*) – Data of shape [n_samples, n_features].

**Returns** Predicted probability values.

**Return type** pd.DataFrame

**save**(*self*, *file_path*, *pickle_protocol=cloudpickle.DEFAULT_PROTOCOL*)

Saves component at file path.

**Parameters**

- **file_path** (*str*) – Location to save file.
- **pickle_protocol** (*int*) – The pickle data stream format.

**update_parameters**(*self*, *update_dict*, *reset_fit=True*)

Updates the parameter dictionary of the component.

**Parameters**

- **update_dict** (*dict*) – A dict of parameters to update.
- **reset_fit** (*bool, optional*) – If True, will set *_is_fitted* to False.

## decision_tree_classifier

Decision Tree Classifier.

### Module Contents

### Classes Summary

| | |
|---|---|
| *DecisionTreeClassifier* | Decision Tree Classifier. |

### Contents

**class** evalml.pipelines.components.estimators.classifiers.decision_tree_classifier.**DecisionTreeClassifie**

Decision Tree Classifier.

**Parameters**

- **criterion** (*{"gini", "entropy"}*) – The function to measure the quality of a split. Supported criteria are "gini" for the Gini impurity and "entropy" for the information gain. Defaults to "gini".
- **max_features** (*int, float or {"auto", "sqrt", "log2"}*) – The number of features to consider when looking for the best split:
  - If int, then consider max_features features at each split.

- If float, then max_features is a fraction and int(max_features * n_features) features are considered at each split.
- If "auto", then max_features=sqrt(n_features).
- If "sqrt", then max_features=sqrt(n_features).
- If "log2", then max_features=log2(n_features).
- If None, then max_features = n_features.

The search for a split does not stop until at least one valid partition of the node samples is found, even if it requires to effectively inspect more than max_features features. Defaults to "auto".

- **max_depth** (*int*) – The maximum depth of the tree. Defaults to 6.
- **min_samples_split** (*int or float*) – The minimum number of samples required to split an internal node:
  - If int, then consider min_samples_split as the minimum number.
  - If float, then min_samples_split is a fraction and ceil(min_samples_split * n_samples) are the minimum number of samples for each split.

  Defaults to 2.
- **min_weight_fraction_leaf** (*float*) – The minimum weighted fraction of the sum total of weights (of all the input samples) required to be at a leaf node. Defaults to 0.0.
- **random_seed** (*int*) – Seed for the random number generator. Defaults to 0.

**Attributes**

| | |
|---|---|
| **hyper-parameter_ranges** | { "criterion": ["gini", "entropy"], "max_features": ["auto", "sqrt", "log2"], "max_depth": Integer(4, 10),} |
| **model_family** | ModelFamily.DECISION_TREE |
| **modifies_features** | True |
| **modifies_target** | False |
| **name** | Decision Tree Classifier |
| **supported_problem_types** | [ ProblemTypes.BINARY, ProblemTypes.MULTICLASS, ProblemTypes.TIME_SERIES_BINARY, ProblemTypes.TIME_SERIES_MULTICLASS,] |
| **training_only** | False |

**Methods**

| | |
|---|---|
| *clone* | Constructs a new component with the same parameters and random state. |
| *default_parameters* | Returns the default parameters for this component. |
| *describe* | Describe a component and its parameters. |
| *feature_importance* | Returns importance associated with each feature. |
| *fit* | Fits estimator to data. |
| *get_prediction_intervals* | Find the prediction intervals using the fitted regressor. |
| *load* | Loads component at file path. |
| *needs_fitting* | Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances. |
| *parameters* | Returns the parameters which were used to initialize the component. |
| *predict* | Make predictions using selected features. |
| *predict_proba* | Make probability estimates for labels. |
| *save* | Saves component at file path. |
| *update_parameters* | Updates the parameter dictionary of the component. |

**clone**(*self*)

Constructs a new component with the same parameters and random state.

**Returns** A new instance of this component with identical parameters and random state.

**default_parameters**(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

**Returns** Default parameters for this component.

**Return type** dict

**describe**(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

**Parameters**

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description as dictionary in the format {"name": name, "parameters": parameters}

**Returns** Returns dictionary if return_dict is True, else None.

**Return type** None or dict

**property feature_importance**(*self*) → pandas.Series

Returns importance associated with each feature.

**Returns** Importance associated with each feature.

**Return type** np.ndarray

**Raises** **MethodPropertyNotFoundError** – If estimator does not have a feature_importance method or a component_obj that implements feature_importance.

**fit**(*self*, *X: pandas.DataFrame*, *y: Optional[pandas.Series] = None*)

Fits estimator to data.

**Parameters**

- **X** (*pd.DataFrame*) – The input training data of shape [n_samples, n_features].
- **y** (*pd.Series, optional*) – The target training data of length [n_samples].

**Returns** self

**get_prediction_intervals**(*self, X: pandas.DataFrame, y: Optional[pandas.Series] = None, coverage: List[float] = None, predictions: pandas.Series = None*) → Dict[str, pandas.Series]

Find the prediction intervals using the fitted regressor.

This function takes the predictions of the fitted estimator and calculates the rolling standard deviation across all predictions using a window size of 5. The lower and upper predictions are determined by taking the percent point (quantile) function of the lower tail probability at each bound multiplied by the rolling standard deviation.

**Parameters**

- **X** (*pd.DataFrame*) – Data of shape [n_samples, n_features].
- **y** (*pd.Series*) – Target data. Ignored.
- **coverage** (*list[float]*) – A list of floats between the values 0 and 1 that the upper and lower bounds of the prediction interval should be calculated for.
- **predictions** (*pd.Series*) – Optional list of predictions to use. If None, will generate predictions using *X*.

**Returns** Prediction intervals, keys are in the format {coverage}_lower or {coverage}_upper.

**Return type** dict

**Raises** **MethodPropertyNotFoundError** – If the estimator does not support Time Series Regression as a problem type.

**static load**(*file_path*)

Loads component at file path.

**Parameters** **file_path** (*str*) – Location to load file.

**Returns** ComponentBase object

**needs_fitting**(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need to be fit or whose fit methods do nothing.

**Returns** True.

**property parameters**(*self*)

Returns the parameters which were used to initialize the component.

**predict**(*self, X: pandas.DataFrame*) → pandas.Series

Make predictions using selected features.

**Parameters** **X** (*pd.DataFrame*) – Data of shape [n_samples, n_features].

**Returns** Predicted values.

**Return type** pd.Series

**Raises** `MethodPropertyNotFoundError` – If estimator does not have a predict method or a component_obj that implements predict.

`predict_proba`(*self*, *X: pandas.DataFrame*) → pandas.Series

Make probability estimates for labels.

**Parameters** `X` (*pd.DataFrame*) – Features.

**Returns** Probability estimates.

**Return type** pd.Series

**Raises** `MethodPropertyNotFoundError` – If estimator does not have a predict_proba method or a component_obj that implements predict_proba.

`save`(*self*, *file_path*, *pickle_protocol=cloudpickle.DEFAULT_PROTOCOL*)

Saves component at file path.

**Parameters**

- `file_path` (*str*) – Location to save file.
- `pickle_protocol` (*int*) – The pickle data stream format.

`update_parameters`(*self*, *update_dict*, *reset_fit=True*)

Updates the parameter dictionary of the component.

**Parameters**

- `update_dict` (*dict*) – A dict of parameters to update.
- `reset_fit` (*bool, optional*) – If True, will set *_is_fitted* to False.

### elasticnet_classifier

Elastic Net Classifier. Uses Logistic Regression with elasticnet penalty as the base estimator.

### Module Contents

### Classes Summary

| | |
|---|---|
| *ElasticNetClassifier* | Elastic Net Classifier. Uses Logistic Regression with elasticnet penalty as the base estimator. |

### Contents

**class** evalml.pipelines.components.estimators.classifiers.elasticnet_classifier.**ElasticNetClassifier**(*penal*
*C=1*
*l1_r*
*mult*
*solve*
*n_jo*

*1*,
*ran-*
*dom*
***kw*

Elastic Net Classifier. Uses Logistic Regression with elasticnet penalty as the base estimator.

**Parameters**

- **penalty** (*{"l1", "l2", "elasticnet", "none"}*) – The norm used in penalization. Defaults to "elasticnet".
- **C** (*float*) – Inverse of regularization strength. Must be a positive float. Defaults to 1.0.
- **l1_ratio** (*float*) – The mixing parameter, with 0 <= l1_ratio <= 1. Only used if penalty='elasticnet'. Setting l1_ratio=0 is equivalent to using penalty='l2', while setting l1_ratio=1 is equivalent to using penalty='l1'. For 0 < l1_ratio <1, the penalty is a combination of L1 and L2. Defaults to 0.15.
- **multi_class** (*{"auto", "ovr", "multinomial"}*) – If the option chosen is "ovr", then a binary problem is fit for each label. For "multinomial" the loss minimised is the multinomial loss fit across the entire probability distribution, even when the data is binary. "multinomial" is unavailable when solver="liblinear". "auto" selects "ovr" if the data is binary, or if solver="liblinear", and otherwise selects "multinomial". Defaults to "auto".
- **solver** (*{"newton-cg", "lbfgs", "liblinear", "sag", "saga"}*) – Algorithm to use in the optimization problem. For small datasets, "liblinear" is a good choice, whereas "sag" and "saga" are faster for large ones. For multiclass problems, only "newton-cg", "sag", "saga" and "lbfgs" handle multinomial loss; "liblinear" is limited to one-versus-rest schemes.
  - "newton-cg", "lbfgs", "sag" and "saga" handle L2 or no penalty
  - "liblinear" and "saga" also handle L1 penalty
  - "saga" also supports "elasticnet" penalty
  - "liblinear" does not support setting penalty='none'

  Defaults to "saga".
- **n_jobs** (*int*) – Number of parallel threads used to run xgboost. Note that creating thread contention will significantly slow down the algorithm. Defaults to -1.
- **random_seed** (*int*) – Seed for the random number generator. Defaults to 0.

**Attributes**

| | |
|---|---|
| **hyper-parameter_ranges** | { "C": Real(0.01, 10), "l1_ratio": Real(0, 1)} |
| **model_family** | ModelFamily.LINEAR_MODEL |
| **modifies_features** | True |
| **modifies_target** | False |
| **name** | Elastic Net Classifier |
| **supported_problem_types** | [ ProblemTypes.BINARY, ProblemTypes.MULTICLASS, ProblemTypes.TIME_SERIES_BINARY, ProblemTypes.TIME_SERIES_MULTICLASS,] |
| **training_only** | False |

**Methods**

| | |
|---|---|
| *clone* | Constructs a new component with the same parameters and random state. |
| *default_parameters* | Returns the default parameters for this component. |
| *describe* | Describe a component and its parameters. |
| *feature_importance* | Feature importance for fitted ElasticNet classifier. |
| *fit* | Fits ElasticNet classifier component to data. |
| *get_prediction_intervals* | Find the prediction intervals using the fitted regressor. |
| *load* | Loads component at file path. |
| *needs_fitting* | Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances. |
| *parameters* | Returns the parameters which were used to initialize the component. |
| *predict* | Make predictions using selected features. |
| *predict_proba* | Make probability estimates for labels. |
| *save* | Saves component at file path. |
| *update_parameters* | Updates the parameter dictionary of the component. |

**clone**(*self*)

Constructs a new component with the same parameters and random state.

**Returns** A new instance of this component with identical parameters and random state.

**default_parameters**(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

**Returns** Default parameters for this component.

**Return type** dict

**describe**(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

**Parameters**

- **print_name** (*bool, optional*) – whether to print name of component

- **return_dict** (*bool, optional*) – whether to return description as dictionary in the format {"name": name, "parameters": parameters}

**Returns** Returns dictionary if return_dict is True, else None.

**Return type** None or dict

**property feature_importance**(*self*)

Feature importance for fitted ElasticNet classifier.

**fit**(*self*, *X*, *y*)

Fits ElasticNet classifier component to data.

**Parameters**

- **X** (*pd.DataFrame*) – The input training data of shape [n_samples, n_features].
- **y** (*pd.Series*) – The target training data of length [n_samples].

**Returns** self

**get_prediction_intervals**(*self*, *X: pandas.DataFrame*, *y: Optional[pandas.Series] = None*, *coverage: List[float] = None*, *predictions: pandas.Series = None*) → Dict[str, pandas.Series]

Find the prediction intervals using the fitted regressor.

This function takes the predictions of the fitted estimator and calculates the rolling standard deviation across all predictions using a window size of 5. The lower and upper predictions are determined by taking the percent point (quantile) function of the lower tail probability at each bound multiplied by the rolling standard deviation.

**Parameters**

- **X** (*pd.DataFrame*) – Data of shape [n_samples, n_features].
- **y** (*pd.Series*) – Target data. Ignored.
- **coverage** (*list[float]*) – A list of floats between the values 0 and 1 that the upper and lower bounds of the prediction interval should be calculated for.
- **predictions** (*pd.Series*) – Optional list of predictions to use. If None, will generate predictions using *X*.

**Returns** Prediction intervals, keys are in the format {coverage}_lower or {coverage}_upper.

**Return type** dict

**Raises** **MethodPropertyNotFoundError** – If the estimator does not support Time Series Regression as a problem type.

**static load**(*file_path*)

Loads component at file path.

**Parameters** **file_path** (*str*) – Location to load file.

**Returns** ComponentBase object

**needs_fitting**(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need to be fit or whose fit methods do nothing.

**Returns** True.

**property parameters**(*self*)

Returns the parameters which were used to initialize the component.

**predict**(*self*, *X: pandas.DataFrame*) → pandas.Series

Make predictions using selected features.

**Parameters** **X** (*pd.DataFrame*) – Data of shape [n_samples, n_features].

**Returns** Predicted values.

**Return type** pd.Series

**Raises** **MethodPropertyNotFoundError** – If estimator does not have a predict method or a component_obj that implements predict.

**predict_proba**(*self*, *X: pandas.DataFrame*) → pandas.Series

Make probability estimates for labels.

**Parameters** **X** (*pd.DataFrame*) – Features.

**Returns** Probability estimates.

**Return type** pd.Series

**Raises** **MethodPropertyNotFoundError** – If estimator does not have a predict_proba method or a component_obj that implements predict_proba.

**save**(*self*, *file_path*, *pickle_protocol=cloudpickle.DEFAULT_PROTOCOL*)

Saves component at file path.

**Parameters**

- **file_path** (*str*) – Location to save file.
- **pickle_protocol** (*int*) – The pickle data stream format.

**update_parameters**(*self*, *update_dict*, *reset_fit=True*)

Updates the parameter dictionary of the component.

**Parameters**

- **update_dict** (*dict*) – A dict of parameters to update.
- **reset_fit** (*bool, optional*) – If True, will set *_is_fitted* to False.

## et_classifier

Extra Trees Classifier.

### Module Contents

### Classes Summary

| | |
|---|---|
| *ExtraTreesClassifier* | Extra Trees Classifier. |

### Contents

**class** evalml.pipelines.components.estimators.classifiers.et_classifier.**ExtraTreesClassifier**(*n_estimators=*
*max_features=*
*max_depth=6,*
*min_samples_s*
*min_weight_fra*
*n_jobs=-*

*1,*
*ran-*
*dom_seed=0,*
***kwargs*)

Extra Trees Classifier.

**Parameters**

- **n_estimators** (*float*) – The number of trees in the forest. Defaults to 100.
- **max_features** (*int, float or {"auto", "sqrt", "log2"}*) – The number of features to consider when looking for the best split:
  - If int, then consider max_features features at each split.
  - If float, then max_features is a fraction and int(max_features * n_features) features are considered at each split.
  - If "auto", then max_features=sqrt(n_features).
  - If "sqrt", then max_features=sqrt(n_features).
  - If "log2", then max_features=log2(n_features).
  - If None, then max_features = n_features.

  The search for a split does not stop until at least one valid partition of the node samples is found, even if it requires to effectively inspect more than max_features features. Defaults to "auto".
- **max_depth** (*int*) – The maximum depth of the tree. Defaults to 6.
- **min_samples_split** (*int or float*) – The minimum number of samples required to split an internal node:
  - If int, then consider min_samples_split as the minimum number.
  - If float, then min_samples_split is a fraction and ceil(min_samples_split * n_samples) are the minimum number of samples for each split.
- **2.** (*Defaults to*) –
- **min_weight_fraction_leaf** (*float*) – The minimum weighted fraction of the sum total of weights (of all the input samples) required to be at a leaf node. Defaults to 0.0.
- **n_jobs** (*int or None*) – Number of jobs to run in parallel. -1 uses all processes. Defaults to -1.
- **random_seed** (*int*) – Seed for the random number generator. Defaults to 0.

**Attributes**

| | |
|---|---|
| **hyper-parameter_ranges** | { "n_estimators": Integer(10, 1000), "max_features": ["auto", "sqrt", "log2"], "max_depth": Integer(4, 10),} |
| **model_family** | ModelFamily.EXTRA_TREES |
| **modifies_features** | True |
| **modifies_target** | False |
| **name** | Extra Trees Classifier |
| **supported_problem_types** | [ ProblemTypes.BINARY, ProblemTypes.MULTICLASS, ProblemTypes.TIME_SERIES_BINARY, ProblemTypes.TIME_SERIES_MULTICLASS,] |
| **training_only** | False |

**Methods**

| | |
|---|---|
| *clone* | Constructs a new component with the same parameters and random state. |
| *default_parameters* | Returns the default parameters for this component. |
| *describe* | Describe a component and its parameters. |
| *feature_importance* | Returns importance associated with each feature. |
| *fit* | Fits estimator to data. |
| *get_prediction_intervals* | Find the prediction intervals using the fitted regressor. |
| *load* | Loads component at file path. |
| *needs_fitting* | Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances. |
| *parameters* | Returns the parameters which were used to initialize the component. |
| *predict* | Make predictions using selected features. |
| *predict_proba* | Make probability estimates for labels. |
| *save* | Saves component at file path. |
| *update_parameters* | Updates the parameter dictionary of the component. |

**clone**(*self*)

Constructs a new component with the same parameters and random state.

**Returns** A new instance of this component with identical parameters and random state.

**default_parameters**(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

**Returns** Default parameters for this component.

**Return type** dict

**describe**(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

**Parameters**

- **print_name** (*bool, optional*) – whether to print name of component

- **return_dict** (*bool, optional*) – whether to return description as dictionary in the format {"name": name, "parameters": parameters}

**Returns** Returns dictionary if return_dict is True, else None.

**Return type** None or dict

**property feature_importance**(*self*) → pandas.Series

Returns importance associated with each feature.

**Returns** Importance associated with each feature.

**Return type** np.ndarray

**Raises** **MethodPropertyNotFoundError** – If estimator does not have a feature_importance method or a component_obj that implements feature_importance.

**fit**(*self, X: pandas.DataFrame, y: Optional[pandas.Series] = None*)

Fits estimator to data.

**Parameters**

- **X** (*pd.DataFrame*) – The input training data of shape [n_samples, n_features].
- **y** (*pd.Series, optional*) – The target training data of length [n_samples].

**Returns** self

**get_prediction_intervals**(*self, X: pandas.DataFrame, y: Optional[pandas.Series] = None, coverage: List[float] = None, predictions: pandas.Series = None*) → Dict[str, pandas.Series]

Find the prediction intervals using the fitted regressor.

This function takes the predictions of the fitted estimator and calculates the rolling standard deviation across all predictions using a window size of 5. The lower and upper predictions are determined by taking the percent point (quantile) function of the lower tail probability at each bound multiplied by the rolling standard deviation.

**Parameters**

- **X** (*pd.DataFrame*) – Data of shape [n_samples, n_features].
- **y** (*pd.Series*) – Target data. Ignored.
- **coverage** (*list[float]*) – A list of floats between the values 0 and 1 that the upper and lower bounds of the prediction interval should be calculated for.
- **predictions** (*pd.Series*) – Optional list of predictions to use. If None, will generate predictions using *X*.

**Returns** Prediction intervals, keys are in the format {coverage}_lower or {coverage}_upper.

**Return type** dict

**Raises** **MethodPropertyNotFoundError** – If the estimator does not support Time Series Regression as a problem type.

**static load**(*file_path*)

Loads component at file path.

**Parameters** **file_path** (*str*) – Location to load file.

**Returns** ComponentBase object

**needs_fitting**(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need to be fit or whose fit methods do nothing.

**Returns** True.

**property parameters**(*self*)

Returns the parameters which were used to initialize the component.

**predict**(*self*, *X: pandas.DataFrame*) → pandas.Series

Make predictions using selected features.

**Parameters** **X** (*pd.DataFrame*) – Data of shape [n_samples, n_features].

**Returns** Predicted values.

**Return type** pd.Series

**Raises** **MethodPropertyNotFoundError** – If estimator does not have a predict method or a component_obj that implements predict.

**predict_proba**(*self*, *X: pandas.DataFrame*) → pandas.Series

Make probability estimates for labels.

**Parameters** **X** (*pd.DataFrame*) – Features.

**Returns** Probability estimates.

**Return type** pd.Series

**Raises** **MethodPropertyNotFoundError** – If estimator does not have a predict_proba method or a component_obj that implements predict_proba.

**save**(*self*, *file_path*, *pickle_protocol=cloudpickle.DEFAULT_PROTOCOL*)

Saves component at file path.

**Parameters**

- **file_path** (*str*) – Location to save file.
- **pickle_protocol** (*int*) – The pickle data stream format.

**update_parameters**(*self*, *update_dict*, *reset_fit=True*)

Updates the parameter dictionary of the component.

**Parameters**

- **update_dict** (*dict*) – A dict of parameters to update.
- **reset_fit** (*bool, optional*) – If True, will set *_is_fitted* to False.

### kneighbors_classifier

K-Nearest Neighbors Classifier.

### Module Contents

### Classes Summary

| | |
|---|---|
| *KNeighborsClassifier* | K-Nearest Neighbors Classifier. |

### Contents

**class** evalml.pipelines.components.estimators.classifiers.kneighbors_classifier.**KNeighborsClassifier**(*n_ne weig al- go- rithn leaf_ p=2 ran- dom **kv*

K-Nearest Neighbors Classifier.

**Parameters**

- **n_neighbors** (*int*) – Number of neighbors to use by default. Defaults to 5.
- **weights** (*{'uniform', 'distance'} or callable*) – Weight function used in prediction. Can be:
  - 'uniform' : uniform weights. All points in each neighborhood are weighted equally.
  - 'distance' : weight points by the inverse of their distance. in this case, closer neighbors of a query point will have a greater influence than neighbors which are further away.
  - [callable] : a user-defined function which accepts an array of distances, and returns an array of the same shape containing the weights.

  Defaults to "uniform".
- **algorithm** (*{'auto', 'ball_tree', 'kd_tree', 'brute'}*) – Algorithm used to compute the nearest neighbors:
  - 'ball_tree' will use BallTree
  - 'kd_tree' will use KDTree
  - 'brute' will use a brute-force search.

  'auto' will attempt to decide the most appropriate algorithm based on the values passed to fit method. Defaults to "auto". Note: fitting on sparse input will override the setting of this parameter, using brute force.
- **leaf_size** (*int*) – Leaf size passed to BallTree or KDTree. This can affect the speed of the construction and query, as well as the memory required to store the tree. The optimal value depends on the nature of the problem. Defaults to 30.

- **p** (*int*) – Power parameter for the Minkowski metric. When p = 1, this is equivalent to using manhattan_distance (l1), and euclidean_distance (l2) for p = 2. For arbitrary p, minkowski_distance (l_p) is used. Defaults to 2.
- **random_seed** (*int*) – Seed for the random number generator. Defaults to 0.

**Attributes**

| | |
|---|---|
| **hyper-parameter_ranges** | { "n_neighbors": Integer(2, 12), "weights": ["uniform", "distance"], "algorithm": ["auto", "ball_tree", "kd_tree", "brute"], "leaf_size": Integer(10, 30), "p": Integer(1, 5),} |
| **model_family** | ModelFamily.K_NEIGHBORS |
| **modifies_features** | True |
| **modifies_target** | False |
| **name** | KNN Classifier |
| **supported_problem_types** | [ ProblemTypes.BINARY, ProblemTypes.MULTICLASS, ProblemTypes.TIME_SERIES_BINARY, ProblemTypes.TIME_SERIES_MULTICLASS,] |
| **training_only** | False |

**Methods**

| | |
|---|---|
| *clone* | Constructs a new component with the same parameters and random state. |
| *default_parameters* | Returns the default parameters for this component. |
| *describe* | Describe a component and its parameters. |
| *feature_importance* | Returns array of 0's matching the input number of features as feature_importance is not defined for KNN classifiers. |
| *fit* | Fits estimator to data. |
| *get_prediction_intervals* | Find the prediction intervals using the fitted regressor. |
| *load* | Loads component at file path. |
| *needs_fitting* | Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances. |
| *parameters* | Returns the parameters which were used to initialize the component. |
| *predict* | Make predictions using selected features. |
| *predict_proba* | Make probability estimates for labels. |
| *save* | Saves component at file path. |
| *update_parameters* | Updates the parameter dictionary of the component. |

**clone**(*self*)

Constructs a new component with the same parameters and random state.

**Returns** A new instance of this component with identical parameters and random state.

**default_parameters**(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

**Returns** Default parameters for this component.

**Return type** dict

`describe`(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

**Parameters**

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description as dictionary in the format {"name": name, "parameters": parameters}

**Returns** Returns dictionary if return_dict is True, else None.

**Return type** None or dict

`property feature_importance`(*self*)

Returns array of 0's matching the input number of features as feature_importance is not defined for KNN classifiers.

`fit`(*self*, *X: pandas.DataFrame*, *y: Optional[pandas.Series] = None*)

Fits estimator to data.

**Parameters**

- **X** (*pd.DataFrame*) – The input training data of shape [n_samples, n_features].
- **y** (*pd.Series, optional*) – The target training data of length [n_samples].

**Returns** self

`get_prediction_intervals`(*self*, *X: pandas.DataFrame*, *y: Optional[pandas.Series] = None*, *coverage: List[float] = None*, *predictions: pandas.Series = None*) → Dict[str, pandas.Series]

Find the prediction intervals using the fitted regressor.

This function takes the predictions of the fitted estimator and calculates the rolling standard deviation across all predictions using a window size of 5. The lower and upper predictions are determined by taking the percent point (quantile) function of the lower tail probability at each bound multiplied by the rolling standard deviation.

**Parameters**

- **X** (*pd.DataFrame*) – Data of shape [n_samples, n_features].
- **y** (*pd.Series*) – Target data. Ignored.
- **coverage** (*list[float]*) – A list of floats between the values 0 and 1 that the upper and lower bounds of the prediction interval should be calculated for.
- **predictions** (*pd.Series*) – Optional list of predictions to use. If None, will generate predictions using *X*.

**Returns** Prediction intervals, keys are in the format {coverage}_lower or {coverage}_upper.

**Return type** dict

**Raises** **MethodPropertyNotFoundError** – If the estimator does not support Time Series Regression as a problem type.

**static load**(*file_path*)

Loads component at file path.

**Parameters** **file_path** (*str*) – Location to load file.

**Returns** ComponentBase object

**needs_fitting**(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need to be fit or whose fit methods do nothing.

**Returns** True.

**property parameters**(*self*)

Returns the parameters which were used to initialize the component.

**predict**(*self*, *X: pandas.DataFrame*) → pandas.Series

Make predictions using selected features.

**Parameters** **X** (*pd.DataFrame*) – Data of shape [n_samples, n_features].

**Returns** Predicted values.

**Return type** pd.Series

**Raises** **MethodPropertyNotFoundError** – If estimator does not have a predict method or a component_obj that implements predict.

**predict_proba**(*self*, *X: pandas.DataFrame*) → pandas.Series

Make probability estimates for labels.

**Parameters** **X** (*pd.DataFrame*) – Features.

**Returns** Probability estimates.

**Return type** pd.Series

**Raises** **MethodPropertyNotFoundError** – If estimator does not have a predict_proba method or a component_obj that implements predict_proba.

**save**(*self*, *file_path*, *pickle_protocol=cloudpickle.DEFAULT_PROTOCOL*)

Saves component at file path.

**Parameters**

- **file_path** (*str*) – Location to save file.
- **pickle_protocol** (*int*) – The pickle data stream format.

**update_parameters**(*self*, *update_dict*, *reset_fit=True*)

Updates the parameter dictionary of the component.

**Parameters**

- **update_dict** (*dict*) – A dict of parameters to update.
- **reset_fit** (*bool, optional*) – If True, will set *_is_fitted* to False.

### lightgbm_classifier

LightGBM Classifier.

### Module Contents

### Classes Summary

| | |
|---|---|
| *LightGBMClassifier* | LightGBM Classifier. |

### Contents

**class** evalml.pipelines.components.estimators.classifiers.lightgbm_classifier.**LightGBMClassifier**(*boosting_ learn- ing_rate= n_estimat max_dept num_leav min_chila bag- ging_frac bag- ging_freq n_jobs=- 1, ran- dom_seed **kwargs*)

LightGBM Classifier.

**Parameters**

- **boosting_type** (*string*) – Type of boosting to use. Defaults to "gbdt". - 'gbdt' uses traditional Gradient Boosting Decision Tree - "dart", uses Dropouts meet Multiple Additive Regression Trees - "goss", uses Gradient-based One-Side Sampling - "rf", uses Random Forest
- **learning_rate** (*float*) – Boosting learning rate. Defaults to 0.1.
- **n_estimators** (*int*) – Number of boosted trees to fit. Defaults to 100.
- **max_depth** (*int*) – Maximum tree depth for base learners, <=0 means no limit. Defaults to 0.
- **num_leaves** (*int*) – Maximum tree leaves for base learners. Defaults to 31.
- **min_child_samples** (*int*) – Minimum number of data needed in a child (leaf). Defaults to 20.
- **bagging_fraction** (*float*) – LightGBM will randomly select a subset of features on each iteration (tree) without resampling if this is smaller than 1.0. For example, if set to 0.8, LightGBM will select 80% of features before training each tree. This can be used to speed up training and deal with overfitting. Defaults to 0.9.

- **bagging_freq** (*int*) – Frequency for bagging. 0 means bagging is disabled. k means perform bagging at every k iteration. Every k-th iteration, LightGBM will randomly select bagging_fraction * 100 % of the data to use for the next k iterations. Defaults to 0.
- **n_jobs** (*int or None*) – Number of threads to run in parallel. -1 uses all threads. Defaults to -1.
- **random_seed** (*int*) – Seed for the random number generator. Defaults to 0.

**Attributes**

| | |
|---|---|
| **hyper-parameter_ranges** | { "learning_rate": Real(0.000001, 1), "boosting_type": ["gbdt", "dart", "goss", "rf"], "n_estimators": Integer(10, 100), "max_depth": Integer(0, 10), "num_leaves": Integer(2, 100), "min_child_samples": Integer(1, 100), "bagging_fraction": Real(0.000001, 1), "bagging_freq": Integer(0, 1),} |
| **model_family** | ModelFamily.LIGHTGBM |
| **modifies_features** | True |
| **modifies_target** | False |
| **name** | LightGBM Classifier |
| **SEED_MAX** | SEED_BOUNDS.max_bound |
| **SEED_MIN** | 0 |
| **supported_problem_types** | [ ProblemTypes.BINARY, ProblemTypes.MULTICLASS, ProblemTypes.TIME_SERIES_BINARY, ProblemTypes.TIME_SERIES_MULTICLASS,] |
| **training_only** | False |

**Methods**

| | |
|---|---|
| *clone* | Constructs a new component with the same parameters and random state. |
| *default_parameters* | Returns the default parameters for this component. |
| *describe* | Describe a component and its parameters. |
| *feature_importance* | Returns importance associated with each feature. |
| *fit* | Fits LightGBM classifier component to data. |
| *get_prediction_intervals* | Find the prediction intervals using the fitted regressor. |
| *load* | Loads component at file path. |
| *needs_fitting* | Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances. |
| *parameters* | Returns the parameters which were used to initialize the component. |
| *predict* | Make predictions using the fitted LightGBM classifier. |
| *predict_proba* | Make prediction probabilities using the fitted LightGBM classifier. |
| *save* | Saves component at file path. |
| *update_parameters* | Updates the parameter dictionary of the component. |

**clone**(*self*)

Constructs a new component with the same parameters and random state.

**Returns** A new instance of this component with identical parameters and random state.

**default_parameters**(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

**Returns** Default parameters for this component.

**Return type** dict

**describe**(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

**Parameters**

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description as dictionary in the format {"name": name, "parameters": parameters}

**Returns** Returns dictionary if return_dict is True, else None.

**Return type** None or dict

**property feature_importance**(*self*) → pandas.Series

Returns importance associated with each feature.

**Returns** Importance associated with each feature.

**Return type** np.ndarray

**Raises** **MethodPropertyNotFoundError** – If estimator does not have a feature_importance method or a component_obj that implements feature_importance.

**fit**(*self*, *X*, *y=None*)

Fits LightGBM classifier component to data.

**Parameters**

- **X** (*pd.DataFrame*) – The input training data of shape [n_samples, n_features].
- **y** (*pd.Series*) – The target training data of length [n_samples].

**Returns** self

**get_prediction_intervals**(*self*, *X: pandas.DataFrame*, *y: Optional[pandas.Series] = None*, *coverage: List[float] = None*, *predictions: pandas.Series = None*) → Dict[str, pandas.Series]

Find the prediction intervals using the fitted regressor.

This function takes the predictions of the fitted estimator and calculates the rolling standard deviation across all predictions using a window size of 5. The lower and upper predictions are determined by taking the percent point (quantile) function of the lower tail probability at each bound multiplied by the rolling standard deviation.

**Parameters**

- **X** (*pd.DataFrame*) – Data of shape [n_samples, n_features].
- **y** (*pd.Series*) – Target data. Ignored.
- **coverage** (*list[float]*) – A list of floats between the values 0 and 1 that the upper and lower bounds of the prediction interval should be calculated for.

- **predictions** (*pd.Series*) – Optional list of predictions to use. If None, will generate predictions using *X*.

**Returns** Prediction intervals, keys are in the format {coverage}_lower or {coverage}_upper.

**Return type** dict

**Raises** **MethodPropertyNotFoundError** – If the estimator does not support Time Series Regression as a problem type.

**static load**(*file_path*)

Loads component at file path.

**Parameters** **file_path** (*str*) – Location to load file.

**Returns** ComponentBase object

**needs_fitting**(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need to be fit or whose fit methods do nothing.

**Returns** True.

**property parameters**(*self*)

Returns the parameters which were used to initialize the component.

**predict**(*self*, *X*)

Make predictions using the fitted LightGBM classifier.

**Parameters** **X** (*pd.DataFrame*) – Data of shape [n_samples, n_features].

**Returns** Predicted values.

**Return type** pd.DataFrame

**predict_proba**(*self*, *X*)

Make prediction probabilities using the fitted LightGBM classifier.

**Parameters** **X** (*pd.DataFrame*) – Data of shape [n_samples, n_features].

**Returns** Predicted probability values.

**Return type** pd.DataFrame

**save**(*self*, *file_path*, *pickle_protocol=cloudpickle.DEFAULT_PROTOCOL*)

Saves component at file path.

**Parameters**

- **file_path** (*str*) – Location to save file.
- **pickle_protocol** (*int*) – The pickle data stream format.

**update_parameters**(*self*, *update_dict*, *reset_fit=True*)

Updates the parameter dictionary of the component.

**Parameters**

- **update_dict** (*dict*) – A dict of parameters to update.
- **reset_fit** (*bool, optional*) – If True, will set *_is_fitted* to False.

### logistic_regression_classifier

Logistic Regression Classifier.

### Module Contents

### Classes Summary

| | |
|---|---|
| *LogisticRegressionClassifier* | Logistic Regression Classifier. |

### Contents

**class** evalml.pipelines.components.estimators.classifiers.logistic_regression_classifier.**LogisticRegressi**

Logistic Regression Classifier.

**Parameters**

- **penalty** (*{"l1", "l2", "elasticnet", "none"}*) – The norm used in penalization. Defaults to "l2".
- **C** (*float*) – Inverse of regularization strength. Must be a positive float. Defaults to 1.0.
- **multi_class** (*{"auto", "ovr", "multinomial"}*) – If the option chosen is "ovr", then a binary problem is fit for each label. For "multinomial" the loss minimised is the multinomial loss fit across the entire probability distribution, even when the data is binary. "multinomial" is unavailable when solver="liblinear". "auto" selects "ovr" if the data is binary, or if solver="liblinear", and otherwise selects "multinomial". Defaults to "auto".
- **solver** (*{"newton-cg", "lbfgs", "liblinear", "sag", "saga"}*) – Algorithm to use in the optimization problem. For small datasets, "liblinear" is a good choice, whereas "sag" and "saga" are faster for large ones. For multiclass problems, only "newton-cg", "sag", "saga" and "lbfgs" handle multinomial loss; "liblinear" is limited to one-versus-rest schemes.
  - "newton-cg", "lbfgs", "sag" and "saga" handle L2 or no penalty
  - "liblinear" and "saga" also handle L1 penalty
  - "saga" also supports "elasticnet" penalty
  - "liblinear" does not support setting penalty='none'

  Defaults to "lbfgs".
- **n_jobs** (*int*) – Number of parallel threads used to run xgboost. Note that creating thread contention will significantly slow down the algorithm. Defaults to -1.
- **random_seed** (*int*) – Seed for the random number generator. Defaults to 0.

**Attributes**

| | |
|---|---|
| **hyper-parameter_ranges** | { "penalty": ["l2"], "C": Real(0.01, 10),} |
| **model_family** | ModelFamily.LINEAR_MODEL |
| **modifies_features** | True |
| **modifies_target** | False |
| **name** | Logistic Regression Classifier |
| **supported_problem_types** | [ ProblemTypes.BINARY, ProblemTypes.MULTICLASS, ProblemTypes.TIME_SERIES_BINARY, ProblemTypes.TIME_SERIES_MULTICLASS,] |
| **training_only** | False |

**Methods**

| | |
|---|---|
| *clone* | Constructs a new component with the same parameters and random state. |
| *default_parameters* | Returns the default parameters for this component. |
| *describe* | Describe a component and its parameters. |
| *feature_importance* | Feature importance for fitted logistic regression classifier. |
| *fit* | Fits estimator to data. |
| *get_prediction_intervals* | Find the prediction intervals using the fitted regressor. |
| *load* | Loads component at file path. |
| *needs_fitting* | Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances. |
| *parameters* | Returns the parameters which were used to initialize the component. |
| *predict* | Make predictions using selected features. |
| *predict_proba* | Make probability estimates for labels. |
| *save* | Saves component at file path. |
| *update_parameters* | Updates the parameter dictionary of the component. |

**clone**(*self*)

Constructs a new component with the same parameters and random state.

**Returns** A new instance of this component with identical parameters and random state.

**default_parameters**(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

**Returns** Default parameters for this component.

**Return type** dict

**describe**(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

**Parameters**

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description as dictionary in the format {"name": name, "parameters": parameters}

**Returns** Returns dictionary if return_dict is True, else None.

**Return type** None or dict

**property feature_importance**(*self*)

Feature importance for fitted logistic regression classifier.

**fit**(*self, X: pandas.DataFrame, y: Optional[pandas.Series] = None*)

Fits estimator to data.

**Parameters**

- **X** (*pd.DataFrame*) – The input training data of shape [n_samples, n_features].
- **y** (*pd.Series, optional*) – The target training data of length [n_samples].

**Returns** self

**get_prediction_intervals**(*self, X: pandas.DataFrame, y: Optional[pandas.Series] = None, coverage: List[float] = None, predictions: pandas.Series = None*) → Dict[str, pandas.Series]

Find the prediction intervals using the fitted regressor.

This function takes the predictions of the fitted estimator and calculates the rolling standard deviation across all predictions using a window size of 5. The lower and upper predictions are determined by taking the percent point (quantile) function of the lower tail probability at each bound multiplied by the rolling standard deviation.

**Parameters**

- **X** (*pd.DataFrame*) – Data of shape [n_samples, n_features].
- **y** (*pd.Series*) – Target data. Ignored.
- **coverage** (*list[float]*) – A list of floats between the values 0 and 1 that the upper and lower bounds of the prediction interval should be calculated for.
- **predictions** (*pd.Series*) – Optional list of predictions to use. If None, will generate predictions using *X*.

**Returns** Prediction intervals, keys are in the format {coverage}_lower or {coverage}_upper.

**Return type** dict

**Raises** **MethodPropertyNotFoundError** – If the estimator does not support Time Series Regression as a problem type.

**static load**(*file_path*)

Loads component at file path.

**Parameters** **file_path** (*str*) – Location to load file.

**Returns** ComponentBase object

**needs_fitting**(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need to be fit or whose fit methods do nothing.

**Returns** True.

**property parameters**(*self*)

Returns the parameters which were used to initialize the component.

**predict**(*self*, *X: pandas.DataFrame*) → pandas.Series

Make predictions using selected features.

**Parameters** **X** (*pd.DataFrame*) – Data of shape [n_samples, n_features].

**Returns** Predicted values.

**Return type** pd.Series

**Raises** **MethodPropertyNotFoundError** – If estimator does not have a predict method or a component_obj that implements predict.

**predict_proba**(*self*, *X: pandas.DataFrame*) → pandas.Series

Make probability estimates for labels.

**Parameters** **X** (*pd.DataFrame*) – Features.

**Returns** Probability estimates.

**Return type** pd.Series

**Raises** **MethodPropertyNotFoundError** – If estimator does not have a predict_proba method or a component_obj that implements predict_proba.

**save**(*self*, *file_path*, *pickle_protocol=cloudpickle.DEFAULT_PROTOCOL*)

Saves component at file path.

**Parameters**

- **file_path** (*str*) – Location to save file.
- **pickle_protocol** (*int*) – The pickle data stream format.

**update_parameters**(*self*, *update_dict*, *reset_fit=True*)

Updates the parameter dictionary of the component.

**Parameters**

- **update_dict** (*dict*) – A dict of parameters to update.
- **reset_fit** (*bool, optional*) – If True, will set *_is_fitted* to False.

### rf_classifier

Random Forest Classifier.

### Module Contents

### Classes Summary

| | |
|---|---|
| *RandomForestClassifier* | Random Forest Classifier. |

### Contents

**class** evalml.pipelines.components.estimators.classifiers.rf_classifier.**RandomForestClassifier**(*n_estimator*
*max_depth=*
*n_jobs=-*

*1*,
*ran-*
*dom_seed=(*
***kwargs*)

Random Forest Classifier.

**Parameters**

- **n_estimators** (*float*) – The number of trees in the forest. Defaults to 100.
- **max_depth** (*int*) – Maximum tree depth for base learners. Defaults to 6.
- **n_jobs** (*int or None*) – Number of jobs to run in parallel. -1 uses all processes. Defaults to -1.
- **random_seed** (*int*) – Seed for the random number generator. Defaults to 0.

**Attributes**

| | |
|---|---|
| **hyper-parameter_ranges** | { "n_estimators": Integer(10, 1000), "max_depth": Integer(1, 10),} |
| **model_family** | ModelFamily.RANDOM_FOREST |
| **modifies_features** | True |
| **modifies_target** | False |
| **name** | Random Forest Classifier |
| **supported_problem_types** | [ ProblemTypes.BINARY, ProblemTypes.MULTICLASS, ProblemTypes.TIME_SERIES_BINARY, ProblemTypes.TIME_SERIES_MULTICLASS,] |
| **training_only** | False |

**Methods**

| | |
|---|---|
| *clone* | Constructs a new component with the same parameters and random state. |
| *default_parameters* | Returns the default parameters for this component. |
| *describe* | Describe a component and its parameters. |
| *feature_importance* | Returns importance associated with each feature. |
| *fit* | Fits estimator to data. |
| *get_prediction_intervals* | Find the prediction intervals using the fitted regressor. |
| *load* | Loads component at file path. |
| *needs_fitting* | Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances. |
| *parameters* | Returns the parameters which were used to initialize the component. |
| *predict* | Make predictions using selected features. |
| *predict_proba* | Make probability estimates for labels. |
| *save* | Saves component at file path. |
| *update_parameters* | Updates the parameter dictionary of the component. |

**clone**(*self*)

Constructs a new component with the same parameters and random state.

**Returns** A new instance of this component with identical parameters and random state.

**default_parameters**(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

**Returns** Default parameters for this component.

**Return type** dict

**describe**(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

**Parameters**

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description as dictionary in the format {"name": name, "parameters": parameters}

**Returns** Returns dictionary if return_dict is True, else None.

**Return type** None or dict

**property feature_importance**(*self*) → pandas.Series

Returns importance associated with each feature.

**Returns** Importance associated with each feature.

**Return type** np.ndarray

**Raises** **MethodPropertyNotFoundError** – If estimator does not have a feature_importance method or a component_obj that implements feature_importance.

**fit**(*self*, *X: pandas.DataFrame*, *y: Optional[pandas.Series] = None*)

Fits estimator to data.

**Parameters**

- **X** (*pd.DataFrame*) – The input training data of shape [n_samples, n_features].
- **y** (*pd.Series, optional*) – The target training data of length [n_samples].

**Returns** self

**get_prediction_intervals**(*self, X: pandas.DataFrame, y: Optional[pandas.Series] = None, coverage: List[float] = None, predictions: pandas.Series = None*) → Dict[str, pandas.Series]

Find the prediction intervals using the fitted regressor.

This function takes the predictions of the fitted estimator and calculates the rolling standard deviation across all predictions using a window size of 5. The lower and upper predictions are determined by taking the percent point (quantile) function of the lower tail probability at each bound multiplied by the rolling standard deviation.

**Parameters**

- **X** (*pd.DataFrame*) – Data of shape [n_samples, n_features].
- **y** (*pd.Series*) – Target data. Ignored.
- **coverage** (*list[float]*) – A list of floats between the values 0 and 1 that the upper and lower bounds of the prediction interval should be calculated for.
- **predictions** (*pd.Series*) – Optional list of predictions to use. If None, will generate predictions using *X*.

**Returns** Prediction intervals, keys are in the format {coverage}_lower or {coverage}_upper.

**Return type** dict

**Raises** **MethodPropertyNotFoundError** – If the estimator does not support Time Series Regression as a problem type.

**static load**(*file_path*)

Loads component at file path.

**Parameters** **file_path** (*str*) – Location to load file.

**Returns** ComponentBase object

**needs_fitting**(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need to be fit or whose fit methods do nothing.

**Returns** True.

**property parameters**(*self*)

Returns the parameters which were used to initialize the component.

**predict**(*self, X: pandas.DataFrame*) → pandas.Series

Make predictions using selected features.

**Parameters** **X** (*pd.DataFrame*) – Data of shape [n_samples, n_features].

**Returns** Predicted values.

**Return type** pd.Series

**Raises** `MethodPropertyNotFoundError` – If estimator does not have a predict method or a component_obj that implements predict.

`predict_proba`(*self*, *X: pandas.DataFrame*) → pandas.Series

Make probability estimates for labels.

**Parameters** `X` (*pd.DataFrame*) – Features.

**Returns** Probability estimates.

**Return type** pd.Series

**Raises** `MethodPropertyNotFoundError` – If estimator does not have a predict_proba method or a component_obj that implements predict_proba.

`save`(*self*, *file_path*, *pickle_protocol=cloudpickle.DEFAULT_PROTOCOL*)

Saves component at file path.

**Parameters**

- `file_path` (*str*) – Location to save file.
- `pickle_protocol` (*int*) – The pickle data stream format.

`update_parameters`(*self*, *update_dict*, *reset_fit=True*)

Updates the parameter dictionary of the component.

**Parameters**

- `update_dict` (*dict*) – A dict of parameters to update.
- `reset_fit` (*bool, optional*) – If True, will set *_is_fitted* to False.

### svm_classifier

Support Vector Machine Classifier.

### Module Contents

### Classes Summary

| | |
|---|---|
| *SVMClassifier* | Support Vector Machine Classifier. |

### Contents

`class evalml.pipelines.components.estimators.classifiers.svm_classifier.SVMClassifier`(*C=1.0, kernel='rbf', gamma='auto', probability=True, random_seed=0, **kwargs*)

Support Vector Machine Classifier.

**Parameters**

- **C** (*float*) – The regularization parameter. The strength of the regularization is inversely proportional to C. Must be strictly positive. The penalty is a squared l2 penalty. Defaults to 1.0.
- **kernel** (*{"poly", "rbf", "sigmoid"}*) – Specifies the kernel type to be used in the algorithm. Defaults to "rbf".
- **gamma** (*{"scale", "auto"} or float*) – Kernel coefficient for "rbf", "poly" and "sigmoid". Defaults to "auto". - If gamma='scale' is passed then it uses 1 / (n_features * X.var()) as value of gamma - If "auto" (default), uses 1 / n_features
- **probability** (*boolean*) – Whether to enable probability estimates. Defaults to True.
- **random_seed** (*int*) – Seed for the random number generator. Defaults to 0.

**Attributes**

| | |
|---|---|
| **hyperparameter_ranges** | { "C": Real(0, 10), "kernel": ["poly", "rbf", "sigmoid"], "gamma": ["scale", "auto"],} |
| **model_family** | ModelFamily.SVM |
| **modifies_features** | True |
| **modifies_target** | False |
| **name** | SVM Classifier |
| **supported_problem_types** | [ ProblemTypes.BINARY, ProblemTypes.MULTICLASS, ProblemTypes.TIME_SERIES_BINARY, ProblemTypes.TIME_SERIES_MULTICLASS,] |
| **training_only** | False |

**Methods**

| | |
|---|---|
| *clone* | Constructs a new component with the same parameters and random state. |
| *default_parameters* | Returns the default parameters for this component. |
| *describe* | Describe a component and its parameters. |
| *feature_importance* | Feature importance only works with linear kernels. |
| *fit* | Fits estimator to data. |
| *get_prediction_intervals* | Find the prediction intervals using the fitted regressor. |
| *load* | Loads component at file path. |
| *needs_fitting* | Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances. |
| *parameters* | Returns the parameters which were used to initialize the component. |
| *predict* | Make predictions using selected features. |
| *predict_proba* | Make probability estimates for labels. |
| *save* | Saves component at file path. |
| *update_parameters* | Updates the parameter dictionary of the component. |

**clone**(*self*)

Constructs a new component with the same parameters and random state.

**Returns** A new instance of this component with identical parameters and random state.

**default_parameters**(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

**Returns** Default parameters for this component.

**Return type** dict

**describe**(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

**Parameters**

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description as dictionary in the format {"name": name, "parameters": parameters}

**Returns** Returns dictionary if return_dict is True, else None.

**Return type** None or dict

**property feature_importance**(*self*)

Feature importance only works with linear kernels.

If the kernel isn't linear, we return a numpy array of zeros.

**Returns** Feature importance of fitted SVM classifier or a numpy array of zeroes if the kernel is not linear.

**fit**(*self*, *X: pandas.DataFrame*, *y: Optional[pandas.Series] = None*)

Fits estimator to data.

**Parameters**

- **X** (*pd.DataFrame*) – The input training data of shape [n_samples, n_features].
- **y** (*pd.Series, optional*) – The target training data of length [n_samples].

**Returns** self

**get_prediction_intervals**(*self*, *X: pandas.DataFrame*, *y: Optional[pandas.Series] = None*, *coverage: List[float] = None*, *predictions: pandas.Series = None*) → Dict[str, pandas.Series]

Find the prediction intervals using the fitted regressor.

This function takes the predictions of the fitted estimator and calculates the rolling standard deviation across all predictions using a window size of 5. The lower and upper predictions are determined by taking the percent point (quantile) function of the lower tail probability at each bound multiplied by the rolling standard deviation.

**Parameters**

- **X** (*pd.DataFrame*) – Data of shape [n_samples, n_features].
- **y** (*pd.Series*) – Target data. Ignored.
- **coverage** (*list[float]*) – A list of floats between the values 0 and 1 that the upper and lower bounds of the prediction interval should be calculated for.

- **predictions** (*pd.Series*) – Optional list of predictions to use. If None, will generate predictions using *X*.

**Returns** Prediction intervals, keys are in the format {coverage}_lower or {coverage}_upper.

**Return type** dict

**Raises** **MethodPropertyNotFoundError** – If the estimator does not support Time Series Regression as a problem type.

**static load**(*file_path*)

Loads component at file path.

**Parameters** **file_path** (*str*) – Location to load file.

**Returns** ComponentBase object

**needs_fitting**(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need to be fit or whose fit methods do nothing.

**Returns** True.

**property parameters**(*self*)

Returns the parameters which were used to initialize the component.

**predict**(*self*, *X: pandas.DataFrame*) → pandas.Series

Make predictions using selected features.

**Parameters** **X** (*pd.DataFrame*) – Data of shape [n_samples, n_features].

**Returns** Predicted values.

**Return type** pd.Series

**Raises** **MethodPropertyNotFoundError** – If estimator does not have a predict method or a component_obj that implements predict.

**predict_proba**(*self*, *X: pandas.DataFrame*) → pandas.Series

Make probability estimates for labels.

**Parameters** **X** (*pd.DataFrame*) – Features.

**Returns** Probability estimates.

**Return type** pd.Series

**Raises** **MethodPropertyNotFoundError** – If estimator does not have a predict_proba method or a component_obj that implements predict_proba.

**save**(*self*, *file_path*, *pickle_protocol=cloudpickle.DEFAULT_PROTOCOL*)

Saves component at file path.

**Parameters**

- **file_path** (*str*) – Location to save file.
- **pickle_protocol** (*int*) – The pickle data stream format.

**update_parameters**(*self*, *update_dict*, *reset_fit=True*)

Updates the parameter dictionary of the component.

**Parameters**

- **update_dict** (*dict*) – A dict of parameters to update.
- **reset_fit** (*bool, optional*) – If True, will set *_is_fitted* to False.

### vowpal_wabbit_classifiers

Vowpal Wabbit Classifiers.

### Module Contents

### Classes Summary

| | |
|---|---|
| *VowpalWabbitBaseClassifier* | Vowpal Wabbit Base Classifier. |
| *VowpalWabbitBinaryClassifier* | Vowpal Wabbit Binary Classifier. |
| *VowpalWabbitMulticlassClassifier* | Vowpal Wabbit Multiclass Classifier. |

### Contents

**class** evalml.pipelines.components.estimators.classifiers.vowpal_wabbit_classifiers.**VowpalWabbitBaseClass**

Vowpal Wabbit Base Classifier.

**Parameters**

- **loss_function** (*str*) – Specifies the loss function to use. One of {"squared", "classic", "hinge", "logistic", "quantile"}. Defaults to "logistic".
- **learning_rate** (*float*) – Boosting learning rate. Defaults to 0.5.
- **decay_learning_rate** (*float*) – Decay factor for learning_rate. Defaults to 1.0.
- **power_t** (*float*) – Power on learning rate decay. Defaults to 0.5.
- **passes** (*int*) – Number of training passes. Defaults to 1.
- **random_seed** (*int*) – Seed for the random number generator. Defaults to 0.

**Attributes**

| | |
|---|---|
| **hyper-parameter_ranges** | None |
| **model_family** | ModelFamily.VOWPAL_WABBIT |
| **modifies_features** | True |
| **modifies_target** | False |
| **training_only** | False |

**Methods**

| | |
|---|---|
| *clone* | Constructs a new component with the same parameters and random state. |
| *default_parameters* | Returns the default parameters for this component. |
| *describe* | Describe a component and its parameters. |
| *feature_importance* | Feature importance for Vowpal Wabbit classifiers. This is not implemented. |
| *fit* | Fits estimator to data. |
| *get_prediction_intervals* | Find the prediction intervals using the fitted regressor. |
| *load* | Loads component at file path. |
| *name* | Returns string name of this component. |
| *needs_fitting* | Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances. |
| *parameters* | Returns the parameters which were used to initialize the component. |
| *predict* | Make predictions using selected features. |
| *predict_proba* | Make probability estimates for labels. |
| *save* | Saves component at file path. |
| *supported_problem_types* | Problem types this estimator supports. |
| *update_parameters* | Updates the parameter dictionary of the component. |

**clone**(*self*)

Constructs a new component with the same parameters and random state.

**Returns** A new instance of this component with identical parameters and random state.

**default_parameters**(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

**Returns** Default parameters for this component.

**Return type** dict

**describe**(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

**Parameters**

- **print_name** (*bool, optional*) – whether to print name of component

- **return_dict** (*bool, optional*) – whether to return description as dictionary in the format {"name": name, "parameters": parameters}

**Returns** Returns dictionary if return_dict is True, else None.

**Return type** None or dict

**property feature_importance**(*self*)

Feature importance for Vowpal Wabbit classifiers. This is not implemented.

**fit**(*self, X: pandas.DataFrame, y: Optional[pandas.Series] = None*)

Fits estimator to data.

**Parameters**

- **X** (*pd.DataFrame*) – The input training data of shape [n_samples, n_features].
- **y** (*pd.Series, optional*) – The target training data of length [n_samples].

**Returns** self

**get_prediction_intervals**(*self, X: pandas.DataFrame, y: Optional[pandas.Series] = None, coverage: List[float] = None, predictions: pandas.Series = None*) → Dict[str, pandas.Series]

Find the prediction intervals using the fitted regressor.

This function takes the predictions of the fitted estimator and calculates the rolling standard deviation across all predictions using a window size of 5. The lower and upper predictions are determined by taking the percent point (quantile) function of the lower tail probability at each bound multiplied by the rolling standard deviation.

**Parameters**

- **X** (*pd.DataFrame*) – Data of shape [n_samples, n_features].
- **y** (*pd.Series*) – Target data. Ignored.
- **coverage** (*list[float]*) – A list of floats between the values 0 and 1 that the upper and lower bounds of the prediction interval should be calculated for.
- **predictions** (*pd.Series*) – Optional list of predictions to use. If None, will generate predictions using *X*.

**Returns** Prediction intervals, keys are in the format {coverage}_lower or {coverage}_upper.

**Return type** dict

**Raises** **MethodPropertyNotFoundError** – If the estimator does not support Time Series Regression as a problem type.

**static load**(*file_path*)

Loads component at file path.

**Parameters** **file_path** (*str*) – Location to load file.

**Returns** ComponentBase object

**property name**(*cls*)

Returns string name of this component.

**needs_fitting**(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need to be fit or whose fit methods do nothing.

**Returns** True.

**property parameters**(*self*)

Returns the parameters which were used to initialize the component.

**predict**(*self*, *X: pandas.DataFrame*) → pandas.Series

Make predictions using selected features.

**Parameters** **X** (*pd.DataFrame*) – Data of shape [n_samples, n_features].

**Returns** Predicted values.

**Return type** pd.Series

**Raises** **MethodPropertyNotFoundError** – If estimator does not have a predict method or a component_obj that implements predict.

**predict_proba**(*self*, *X: pandas.DataFrame*) → pandas.Series

Make probability estimates for labels.

**Parameters** **X** (*pd.DataFrame*) – Features.

**Returns** Probability estimates.

**Return type** pd.Series

**Raises** **MethodPropertyNotFoundError** – If estimator does not have a predict_proba method or a component_obj that implements predict_proba.

**save**(*self*, *file_path*, *pickle_protocol=cloudpickle.DEFAULT_PROTOCOL*)

Saves component at file path.

**Parameters**

- **file_path** (*str*) – Location to save file.
- **pickle_protocol** (*int*) – The pickle data stream format.

**property supported_problem_types**(*cls*)

Problem types this estimator supports.

**update_parameters**(*self*, *update_dict*, *reset_fit=True*)

Updates the parameter dictionary of the component.

**Parameters**

- **update_dict** (*dict*) – A dict of parameters to update.
- **reset_fit** (*bool, optional*) – If True, will set *_is_fitted* to False.

**class** evalml.pipelines.components.estimators.classifiers.vowpal_wabbit_classifiers.**VowpalWabbitBinaryCla**

Vowpal Wabbit Binary Classifier.

**Parameters**

- **loss_function** (*str*) – Specifies the loss function to use. One of {"squared", "classic", "hinge", "logistic", "quantile"}. Defaults to "logistic".
- **learning_rate** (*float*) – Boosting learning rate. Defaults to 0.5.
- **decay_learning_rate** (*float*) – Decay factor for learning_rate. Defaults to 1.0.
- **power_t** (*float*) – Power on learning rate decay. Defaults to 0.5.
- **passes** (*int*) – Number of training passes. Defaults to 1.
- **random_seed** (*int*) – Seed for the random number generator. Defaults to 0.

**Attributes**

| | |
|---|---|
| **hyper-parameter_ranges** | None |
| **model_family** | ModelFamily.VOWPAL_WABBIT |
| **modifies_features** | True |
| **modifies_target** | False |
| **name** | Vowpal Wabbit Binary Classifier |
| **supported_problem_types** | [ ProblemTypes.BINARY, ProblemTypes.TIME_SERIES_BINARY,] |
| **training_only** | False |

**Methods**

| | |
|---|---|
| *clone* | Constructs a new component with the same parameters and random state. |
| *default_parameters* | Returns the default parameters for this component. |
| *describe* | Describe a component and its parameters. |
| *feature_importance* | Feature importance for Vowpal Wabbit classifiers. This is not implemented. |
| *fit* | Fits estimator to data. |
| *get_prediction_intervals* | Find the prediction intervals using the fitted regressor. |
| *load* | Loads component at file path. |
| *needs_fitting* | Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances. |
| *parameters* | Returns the parameters which were used to initialize the component. |
| *predict* | Make predictions using selected features. |
| *predict_proba* | Make probability estimates for labels. |
| *save* | Saves component at file path. |
| *update_parameters* | Updates the parameter dictionary of the component. |

**clone**(*self*)

Constructs a new component with the same parameters and random state.

**Returns** A new instance of this component with identical parameters and random state.

**default_parameters**(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

**Returns** Default parameters for this component.

**Return type** dict

**describe**(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

**Parameters**

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description as dictionary in the format {"name": name, "parameters": parameters}

**Returns** Returns dictionary if return_dict is True, else None.

**Return type** None or dict

**property feature_importance**(*self*)

Feature importance for Vowpal Wabbit classifiers. This is not implemented.

**fit**(*self*, *X: pandas.DataFrame*, *y: Optional[pandas.Series] = None*)

Fits estimator to data.

**Parameters**

- **X** (*pd.DataFrame*) – The input training data of shape [n_samples, n_features].
- **y** (*pd.Series, optional*) – The target training data of length [n_samples].

**Returns** self

**get_prediction_intervals**(*self*, *X: pandas.DataFrame*, *y: Optional[pandas.Series] = None*, *coverage: List[float] = None*, *predictions: pandas.Series = None*) → Dict[str, pandas.Series]

Find the prediction intervals using the fitted regressor.

This function takes the predictions of the fitted estimator and calculates the rolling standard deviation across all predictions using a window size of 5. The lower and upper predictions are determined by taking the percent point (quantile) function of the lower tail probability at each bound multiplied by the rolling standard deviation.

**Parameters**

- **X** (*pd.DataFrame*) – Data of shape [n_samples, n_features].
- **y** (*pd.Series*) – Target data. Ignored.
- **coverage** (*list[float]*) – A list of floats between the values 0 and 1 that the upper and lower bounds of the prediction interval should be calculated for.
- **predictions** (*pd.Series*) – Optional list of predictions to use. If None, will generate predictions using *X*.

**Returns** Prediction intervals, keys are in the format {coverage}_lower or {coverage}_upper.

**Return type** dict

**Raises** **MethodPropertyNotFoundError** – If the estimator does not support Time Series Regression as a problem type.

**static load**(*file_path*)

Loads component at file path.

**Parameters** **file_path** (*str*) – Location to load file.

**Returns** ComponentBase object

**needs_fitting**(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need to be fit or whose fit methods do nothing.

**Returns** True.

**property parameters**(*self*)

Returns the parameters which were used to initialize the component.

**predict**(*self*, *X: pandas.DataFrame*) → pandas.Series

Make predictions using selected features.

**Parameters** **X** (*pd.DataFrame*) – Data of shape [n_samples, n_features].

**Returns** Predicted values.

**Return type** pd.Series

**Raises** **MethodPropertyNotFoundError** – If estimator does not have a predict method or a component_obj that implements predict.

**predict_proba**(*self*, *X: pandas.DataFrame*) → pandas.Series

Make probability estimates for labels.

**Parameters** **X** (*pd.DataFrame*) – Features.

**Returns** Probability estimates.

**Return type** pd.Series

**Raises** **MethodPropertyNotFoundError** – If estimator does not have a predict_proba method or a component_obj that implements predict_proba.

**save**(*self*, *file_path*, *pickle_protocol=cloudpickle.DEFAULT_PROTOCOL*)

Saves component at file path.

**Parameters**

- **file_path** (*str*) – Location to save file.
- **pickle_protocol** (*int*) – The pickle data stream format.

**update_parameters**(*self*, *update_dict*, *reset_fit=True*)

Updates the parameter dictionary of the component.

**Parameters**

- **update_dict** (*dict*) – A dict of parameters to update.
- **reset_fit** (*bool, optional*) – If True, will set *_is_fitted* to False.

**class** evalml.pipelines.components.estimators.classifiers.vowpal_wabbit_classifiers.**VowpalWabbitMulticlas**

Vowpal Wabbit Multiclass Classifier.

**Parameters**

- **loss_function** (*str*) – Specifies the loss function to use. One of {"squared", "classic", "hinge", "logistic", "quantile"}. Defaults to "logistic".
- **learning_rate** (*float*) – Boosting learning rate. Defaults to 0.5.
- **decay_learning_rate** (*float*) – Decay factor for learning_rate. Defaults to 1.0.
- **power_t** (*float*) – Power on learning rate decay. Defaults to 0.5.
- **random_seed** (*int*) – Seed for the random number generator. Defaults to 0.

**Attributes**

| | |
|---|---|
| **hyper-parameter_ranges** | None |
| **model_family** | ModelFamily.VOWPAL_WABBIT |
| **modifies_features** | True |
| **modifies_target** | False |
| **name** | Vowpal Wabbit Multiclass Classifier |
| **supported_problem_types** | [ ProblemTypes.MULTICLASS, ProblemTypes.TIME_SERIES_MULTICLASS,] |
| **training_only** | False |

**Methods**

| | |
|---|---|
| *clone* | Constructs a new component with the same parameters and random state. |
| *default_parameters* | Returns the default parameters for this component. |
| *describe* | Describe a component and its parameters. |
| *feature_importance* | Feature importance for Vowpal Wabbit classifiers. This is not implemented. |
| *fit* | Fits estimator to data. |
| *get_prediction_intervals* | Find the prediction intervals using the fitted regressor. |
| *load* | Loads component at file path. |
| *needs_fitting* | Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances. |
| *parameters* | Returns the parameters which were used to initialize the component. |
| *predict* | Make predictions using selected features. |
| *predict_proba* | Make probability estimates for labels. |
| *save* | Saves component at file path. |
| *update_parameters* | Updates the parameter dictionary of the component. |

**clone**(*self*)

Constructs a new component with the same parameters and random state.

**Returns** A new instance of this component with identical parameters and random state.

**default_parameters**(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

**Returns** Default parameters for this component.

**Return type** dict

**describe**(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

**Parameters**

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description as dictionary in the format {"name": name, "parameters": parameters}

**Returns** Returns dictionary if return_dict is True, else None.

**Return type** None or dict

**property feature_importance**(*self*)

Feature importance for Vowpal Wabbit classifiers. This is not implemented.

**fit**(*self*, *X: pandas.DataFrame*, *y: Optional[pandas.Series] = None*)

Fits estimator to data.

**Parameters**

- **X** (*pd.DataFrame*) – The input training data of shape [n_samples, n_features].
- **y** (*pd.Series, optional*) – The target training data of length [n_samples].

**Returns** self

**get_prediction_intervals**(*self*, *X: pandas.DataFrame*, *y: Optional[pandas.Series] = None*, *coverage: List[float] = None*, *predictions: pandas.Series = None*) → Dict[str, pandas.Series]

Find the prediction intervals using the fitted regressor.

This function takes the predictions of the fitted estimator and calculates the rolling standard deviation across all predictions using a window size of 5. The lower and upper predictions are determined by taking the percent point (quantile) function of the lower tail probability at each bound multiplied by the rolling standard deviation.

**Parameters**

- **X** (*pd.DataFrame*) – Data of shape [n_samples, n_features].
- **y** (*pd.Series*) – Target data. Ignored.
- **coverage** (*list[float]*) – A list of floats between the values 0 and 1 that the upper and lower bounds of the prediction interval should be calculated for.
- **predictions** (*pd.Series*) – Optional list of predictions to use. If None, will generate predictions using *X*.

**Returns** Prediction intervals, keys are in the format {coverage}_lower or {coverage}_upper.

**Return type** dict

**Raises** **MethodPropertyNotFoundError** – If the estimator does not support Time Series Regression as a problem type.

**static load**(*file_path*)

Loads component at file path.

**Parameters** **file_path** (*str*) – Location to load file.

**Returns** ComponentBase object

**needs_fitting**(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need to be fit or whose fit methods do nothing.

**Returns** True.

**property parameters**(*self*)

Returns the parameters which were used to initialize the component.

**predict**(*self*, *X: pandas.DataFrame*) → pandas.Series

Make predictions using selected features.

**Parameters** **X** (*pd.DataFrame*) – Data of shape [n_samples, n_features].

**Returns** Predicted values.

**Return type** pd.Series

**Raises** **MethodPropertyNotFoundError** – If estimator does not have a predict method or a component_obj that implements predict.

**predict_proba**(*self*, *X: pandas.DataFrame*) → pandas.Series

Make probability estimates for labels.

**Parameters** **X** (*pd.DataFrame*) – Features.

**Returns** Probability estimates.

**Return type** pd.Series

**Raises** **MethodPropertyNotFoundError** – If estimator does not have a predict_proba method or a component_obj that implements predict_proba.

**save**(*self*, *file_path*, *pickle_protocol=cloudpickle.DEFAULT_PROTOCOL*)

Saves component at file path.

**Parameters**

- **file_path** (*str*) – Location to save file.
- **pickle_protocol** (*int*) – The pickle data stream format.

**update_parameters**(*self*, *update_dict*, *reset_fit=True*)

Updates the parameter dictionary of the component.

**Parameters**

- **update_dict** (*dict*) – A dict of parameters to update.
- **reset_fit** (*bool, optional*) – If True, will set *_is_fitted* to False.

## xgboost_classifier

XGBoost Classifier.

### Module Contents

### Classes Summary

| | |
|---|---|
| *XGBoostClassifier* | XGBoost Classifier. |

### Contents

**class** evalml.pipelines.components.estimators.classifiers.xgboost_classifier.**XGBoostClassifier**(*eta=0.1, max_depth= min_child_w n_estimator ran- dom_seed=( eval_metric= n_jobs=12, **kwargs*)

XGBoost Classifier.

**Parameters**

- **eta** (*float*) – Boosting learning rate. Defaults to 0.1.

- **max_depth** (*int*) – Maximum tree depth for base learners. Defaults to 6.
- **min_child_weight** (*float*) – Minimum sum of instance weight (hessian) needed in a child. Defaults to 1.0
- **n_estimators** (*int*) – Number of gradient boosted trees. Equivalent to number of boosting rounds. Defaults to 100.
- **random_seed** (*int*) – Seed for the random number generator. Defaults to 0.
- **n_jobs** (*int*) – Number of parallel threads used to run xgboost. Note that creating thread contention will significantly slow down the algorithm. Defaults to 12.

**Attributes**

| | |
|---|---|
| **hyper-parameter_ranges** | { "eta": Real(0.000001, 1), "max_depth": Integer(1, 10), "min_child_weight": Real(1, 10), "n_estimators": Integer(1, 1000),} |
| **model_family** | ModelFamily.XGBOOST |
| **modifies_features** | True |
| **modifies_target** | False |
| **name** | XGBoost Classifier |
| **SEED_MAX** | None |
| **SEED_MIN** | None |
| **supported_problem_types** | [ ProblemTypes.BINARY, ProblemTypes.MULTICLASS, ProblemTypes.TIME_SERIES_BINARY, ProblemTypes.TIME_SERIES_MULTICLASS,] |
| **training_only** | False |

**Methods**

| | |
|---|---|
| *clone* | Constructs a new component with the same parameters and random state. |
| *default_parameters* | Returns the default parameters for this component. |
| *describe* | Describe a component and its parameters. |
| *feature_importance* | Feature importance of fitted XGBoost classifier. |
| *fit* | Fits XGBoost classifier component to data. |
| *get_prediction_intervals* | Find the prediction intervals using the fitted regressor. |
| *load* | Loads component at file path. |
| *needs_fitting* | Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances. |
| *parameters* | Returns the parameters which were used to initialize the component. |
| *predict* | Make predictions using the fitted XGBoost classifier. |
| *predict_proba* | Make predictions using the fitted CatBoost classifier. |
| *save* | Saves component at file path. |
| *update_parameters* | Updates the parameter dictionary of the component. |

**clone**(*self*)

Constructs a new component with the same parameters and random state.

**Returns** A new instance of this component with identical parameters and random state.

**default_parameters**(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

**Returns** Default parameters for this component.

**Return type** dict

**describe**(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

**Parameters**

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description as dictionary in the format {"name": name, "parameters": parameters}

**Returns** Returns dictionary if return_dict is True, else None.

**Return type** None or dict

**property feature_importance**(*self*)

Feature importance of fitted XGBoost classifier.

**fit**(*self*, *X*, *y=None*)

Fits XGBoost classifier component to data.

**Parameters**

- **X** (*pd.DataFrame*) – The input training data of shape [n_samples, n_features].
- **y** (*pd.Series*) – The target training data of length [n_samples].

**Returns** self

**get_prediction_intervals**(*self*, *X: pandas.DataFrame*, *y: Optional[pandas.Series] = None*, *coverage: List[float] = None*, *predictions: pandas.Series = None*) → Dict[str, pandas.Series]

Find the prediction intervals using the fitted regressor.

This function takes the predictions of the fitted estimator and calculates the rolling standard deviation across all predictions using a window size of 5. The lower and upper predictions are determined by taking the percent point (quantile) function of the lower tail probability at each bound multiplied by the rolling standard deviation.

**Parameters**

- **X** (*pd.DataFrame*) – Data of shape [n_samples, n_features].
- **y** (*pd.Series*) – Target data. Ignored.
- **coverage** (*list[float]*) – A list of floats between the values 0 and 1 that the upper and lower bounds of the prediction interval should be calculated for.
- **predictions** (*pd.Series*) – Optional list of predictions to use. If None, will generate predictions using *X*.

**Returns** Prediction intervals, keys are in the format {coverage}_lower or {coverage}_upper.

**Return type** dict

**Raises** `MethodPropertyNotFoundError` – If the estimator does not support Time Series Regression as a problem type.

`static load`(*file_path*)

Loads component at file path.

**Parameters** `file_path` (*str*) – Location to load file.

**Returns** ComponentBase object

`needs_fitting`(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need to be fit or whose fit methods do nothing.

**Returns** True.

`property parameters`(*self*)

Returns the parameters which were used to initialize the component.

`predict`(*self*, *X*)

Make predictions using the fitted XGBoost classifier.

**Parameters** `X` (*pd.DataFrame*) – Data of shape [n_samples, n_features].

**Returns** Predicted values.

**Return type** pd.DataFrame

`predict_proba`(*self*, *X*)

Make predictions using the fitted CatBoost classifier.

**Parameters** `X` (*pd.DataFrame*) – Data of shape [n_samples, n_features].

**Returns** Predicted values.

**Return type** pd.DataFrame

`save`(*self*, *file_path*, *pickle_protocol=cloudpickle.DEFAULT_PROTOCOL*)

Saves component at file path.

**Parameters**

- `file_path` (*str*) – Location to save file.
- `pickle_protocol` (*int*) – The pickle data stream format.

`update_parameters`(*self*, *update_dict*, *reset_fit=True*)

Updates the parameter dictionary of the component.

**Parameters**

- `update_dict` (*dict*) – A dict of parameters to update.
- `reset_fit` (*bool, optional*) – If True, will set *_is_fitted* to False.

### Package Contents

### Classes Summary

| | |
|---|---|
| *BaselineClassifier* | Classifier that predicts using the specified strategy. |
| *CatBoostClassifier* | CatBoost Classifier, a classifier that uses gradient-boosting on decision trees. CatBoost is an open-source library and natively supports categorical features. |
| *DecisionTreeClassifier* | Decision Tree Classifier. |
| *ElasticNetClassifier* | Elastic Net Classifier. Uses Logistic Regression with elasticnet penalty as the base estimator. |
| *ExtraTreesClassifier* | Extra Trees Classifier. |
| *KNeighborsClassifier* | K-Nearest Neighbors Classifier. |
| *LightGBMClassifier* | LightGBM Classifier. |
| *LogisticRegressionClassifier* | Logistic Regression Classifier. |
| *RandomForestClassifier* | Random Forest Classifier. |
| *SVMClassifier* | Support Vector Machine Classifier. |
| *VowpalWabbitBinaryClassifier* | Vowpal Wabbit Binary Classifier. |
| *VowpalWabbitMulticlassClassifier* | Vowpal Wabbit Multiclass Classifier. |
| *XGBoostClassifier* | XGBoost Classifier. |

### Contents

**class** evalml.pipelines.components.estimators.classifiers.**BaselineClassifier**(*strategy='mode', random_seed=0, **kwargs*)

Classifier that predicts using the specified strategy.

This is useful as a simple baseline classifier to compare with other classifiers.

**Parameters**

- **strategy** (*str*) – Method used to predict. Valid options are "mode", "random" and "random_weighted". Defaults to "mode".
- **random_seed** (*int*) – Seed for the random number generator. Defaults to 0.

**Attributes**

| | |
|---|---|
| **hyper-parameter_ranges** | {} |
| **model_family** | ModelFamily.BASELINE |
| **modifies_features** | True |
| **modifies_target** | False |
| **name** | Baseline Classifier |
| **supported_problem_types** | [ProblemTypes.BINARY, ProblemTypes.MULTICLASS] |
| **training_only** | False |

**Methods**

| | |
|---|---|
| *classes_* | Returns class labels. Will return None before fitting. |
| *clone* | Constructs a new component with the same parameters and random state. |
| *default_parameters* | Returns the default parameters for this component. |
| *describe* | Describe a component and its parameters. |
| *feature_importance* | Returns importance associated with each feature. Since baseline classifiers do not use input features to calculate predictions, returns an array of zeroes. |
| *fit* | Fits baseline classifier component to data. |
| *get_prediction_intervals* | Find the prediction intervals using the fitted regressor. |
| *load* | Loads component at file path. |
| *needs_fitting* | Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances. |
| *parameters* | Returns the parameters which were used to initialize the component. |
| *predict* | Make predictions using the baseline classification strategy. |
| *predict_proba* | Make prediction probabilities using the baseline classification strategy. |
| *save* | Saves component at file path. |
| *update_parameters* | Updates the parameter dictionary of the component. |

**property classes_**(*self*)

Returns class labels. Will return None before fitting.

**Returns** Class names

**Return type** list[str] or list(float)

**clone**(*self*)

Constructs a new component with the same parameters and random state.

**Returns** A new instance of this component with identical parameters and random state.

**default_parameters**(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

**Returns** Default parameters for this component.

**Return type** dict

**describe**(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

**Parameters**

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description as dictionary in the format {"name": name, "parameters": parameters}

**Returns** Returns dictionary if return_dict is True, else None.

**Return type** None or dict

**property feature_importance**(*self*)

Returns importance associated with each feature. Since baseline classifiers do not use input features to calculate predictions, returns an array of zeroes.

**Returns** An array of zeroes

**Return type** pd.Series

**fit**(*self*, *X*, *y=None*)

Fits baseline classifier component to data.

**Parameters**

- **X** (*pd.DataFrame*) – The input training data of shape [n_samples, n_features].
- **y** (*pd.Series*) – The target training data of length [n_samples].

**Returns** self

**Raises** **ValueError** – If y is None.

**get_prediction_intervals**(*self*, *X: pandas.DataFrame*, *y: Optional[pandas.Series] = None*, *coverage: List[float] = None*, *predictions: pandas.Series = None*) → Dict[str, pandas.Series]

Find the prediction intervals using the fitted regressor.

This function takes the predictions of the fitted estimator and calculates the rolling standard deviation across all predictions using a window size of 5. The lower and upper predictions are determined by taking the percent point (quantile) function of the lower tail probability at each bound multiplied by the rolling standard deviation.

**Parameters**

- **X** (*pd.DataFrame*) – Data of shape [n_samples, n_features].
- **y** (*pd.Series*) – Target data. Ignored.
- **coverage** (*list[float]*) – A list of floats between the values 0 and 1 that the upper and lower bounds of the prediction interval should be calculated for.
- **predictions** (*pd.Series*) – Optional list of predictions to use. If None, will generate predictions using *X*.

**Returns** Prediction intervals, keys are in the format {coverage}_lower or {coverage}_upper.

**Return type** dict

**Raises** **MethodPropertyNotFoundError** – If the estimator does not support Time Series Regression as a problem type.

**static load**(*file_path*)

Loads component at file path.

**Parameters** **file_path** (*str*) – Location to load file.

**Returns** ComponentBase object

**needs_fitting**(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need to be fit or whose fit methods do nothing.

**Returns** True.

**property parameters**(*self*)

Returns the parameters which were used to initialize the component.

**predict**(*self*, *X*)

Make predictions using the baseline classification strategy.

**Parameters** **X** (*pd.DataFrame*) – Data of shape [n_samples, n_features].

**Returns** Predicted values.

**Return type** pd.Series

**predict_proba**(*self*, *X*)

Make prediction probabilities using the baseline classification strategy.

**Parameters** **X** (*pd.DataFrame*) – Data of shape [n_samples, n_features].

**Returns** Predicted probability values.

**Return type** pd.DataFrame

**save**(*self*, *file_path*, *pickle_protocol=cloudpickle.DEFAULT_PROTOCOL*)

Saves component at file path.

**Parameters**

- **file_path** (*str*) – Location to save file.
- **pickle_protocol** (*int*) – The pickle data stream format.

**update_parameters**(*self*, *update_dict*, *reset_fit=True*)

Updates the parameter dictionary of the component.

**Parameters**

- **update_dict** (*dict*) – A dict of parameters to update.
- **reset_fit** (*bool, optional*) – If True, will set *_is_fitted* to False.

**class** evalml.pipelines.components.estimators.classifiers.**CatBoostClassifier**(*n_estimators=10*, *eta=0.03*, *max_depth=6*, *bootstrap_type=None*, *silent=True*, *allow_writing_files=False*, *random_seed=0*, *n_jobs=- 1*, ***kwargs*)

CatBoost Classifier, a classifier that uses gradient-boosting on decision trees. CatBoost is an open-source library and natively supports categorical features.

For more information, check out https://catboost.ai/

**Parameters**

- **n_estimators** (*float*) – The maximum number of trees to build. Defaults to 10.
- **eta** (*float*) – The learning rate. Defaults to 0.03.
- **max_depth** (*int*) – The maximum tree depth for base learners. Defaults to 6.

- **bootstrap_type** (*string*) – Defines the method for sampling the weights of objects. Available methods are 'Bayesian', 'Bernoulli', 'MVS'. Defaults to None.
- **silent** (*boolean*) – Whether to use the "silent" logging mode. Defaults to True.
- **allow_writing_files** (*boolean*) – Whether to allow writing snapshot files while training. Defaults to False.
- **n_jobs** (*int or None*) – Number of jobs to run in parallel. -1 uses all processes. Defaults to -1.
- **random_seed** (*int*) – Seed for the random number generator. Defaults to 0.

**Attributes**

| | |
|---|---|
| **hyper-parameter_ranges** | { "n_estimators": Integer(4, 100), "eta": Real(0.000001, 1), "max_depth": Integer(4, 10),} |
| **model_family** | ModelFamily.CATBOOST |
| **modifies_features** | True |
| **modifies_target** | False |
| **name** | CatBoost Classifier |
| **supported_problem_types** | [ ProblemTypes.BINARY, ProblemTypes.MULTICLASS, ProblemTypes.TIME_SERIES_BINARY, ProblemTypes.TIME_SERIES_MULTICLASS,] |
| **training_only** | False |

**Methods**

| | |
|---|---|
| *clone* | Constructs a new component with the same parameters and random state. |
| *default_parameters* | Returns the default parameters for this component. |
| *describe* | Describe a component and its parameters. |
| *feature_importance* | Feature importance of fitted CatBoost classifier. |
| *fit* | Fits CatBoost classifier component to data. |
| *get_prediction_intervals* | Find the prediction intervals using the fitted regressor. |
| *load* | Loads component at file path. |
| *needs_fitting* | Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances. |
| *parameters* | Returns the parameters which were used to initialize the component. |
| *predict* | Make predictions using the fitted CatBoost classifier. |
| *predict_proba* | Make prediction probabilities using the fitted CatBoost classifier. |
| *save* | Saves component at file path. |
| *update_parameters* | Updates the parameter dictionary of the component. |

**clone**(*self*)

Constructs a new component with the same parameters and random state.

**Returns** A new instance of this component with identical parameters and random state.

**default_parameters**(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

**Returns** Default parameters for this component.

**Return type** dict

**describe**(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

**Parameters**

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description as dictionary in the format {"name": name, "parameters": parameters}

**Returns** Returns dictionary if return_dict is True, else None.

**Return type** None or dict

**property feature_importance**(*self*)

Feature importance of fitted CatBoost classifier.

**fit**(*self*, *X*, *y=None*)

Fits CatBoost classifier component to data.

**Parameters**

- **X** (*pd.DataFrame*) – The input training data of shape [n_samples, n_features].
- **y** (*pd.Series*) – The target training data of length [n_samples].

**Returns** self

**get_prediction_intervals**(*self*, *X: pandas.DataFrame*, *y: Optional[pandas.Series] = None*, *coverage: List[float] = None*, *predictions: pandas.Series = None*) → Dict[str, pandas.Series]

Find the prediction intervals using the fitted regressor.

This function takes the predictions of the fitted estimator and calculates the rolling standard deviation across all predictions using a window size of 5. The lower and upper predictions are determined by taking the percent point (quantile) function of the lower tail probability at each bound multiplied by the rolling standard deviation.

**Parameters**

- **X** (*pd.DataFrame*) – Data of shape [n_samples, n_features].
- **y** (*pd.Series*) – Target data. Ignored.
- **coverage** (*list[float]*) – A list of floats between the values 0 and 1 that the upper and lower bounds of the prediction interval should be calculated for.
- **predictions** (*pd.Series*) – Optional list of predictions to use. If None, will generate predictions using *X*.

**Returns** Prediction intervals, keys are in the format {coverage}_lower or {coverage}_upper.

**Return type** dict

**Raises** **MethodPropertyNotFoundError** – If the estimator does not support Time Series Regression as a problem type.

**static load**(*file_path*)

Loads component at file path.

**Parameters** **file_path** (*str*) – Location to load file.

**Returns** ComponentBase object

**needs_fitting**(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need to be fit or whose fit methods do nothing.

**Returns** True.

**property parameters**(*self*)

Returns the parameters which were used to initialize the component.

**predict**(*self*, *X*)

Make predictions using the fitted CatBoost classifier.

**Parameters** **X** (*pd.DataFrame*) – Data of shape [n_samples, n_features].

**Returns** Predicted values.

**Return type** pd.Series

**predict_proba**(*self*, *X*)

Make prediction probabilities using the fitted CatBoost classifier.

**Parameters** **X** (*pd.DataFrame*) – Data of shape [n_samples, n_features].

**Returns** Predicted probability values.

**Return type** pd.DataFrame

**save**(*self*, *file_path*, *pickle_protocol=cloudpickle.DEFAULT_PROTOCOL*)

Saves component at file path.

**Parameters**

- **file_path** (*str*) – Location to save file.
- **pickle_protocol** (*int*) – The pickle data stream format.

**update_parameters**(*self*, *update_dict*, *reset_fit=True*)

Updates the parameter dictionary of the component.

**Parameters**

- **update_dict** (*dict*) – A dict of parameters to update.
- **reset_fit** (*bool, optional*) – If True, will set *_is_fitted* to False.

**class** evalml.pipelines.components.estimators.classifiers.**DecisionTreeClassifier**(*criterion='gini', max_features='auto', max_depth=6, min_samples_split=2, min_weight_fraction_leaf=0.0 random_seed=0, **kwargs*)

Decision Tree Classifier.

**Parameters**

- **criterion** (*{"gini", "entropy"}*) – The function to measure the quality of a split. Supported criteria are "gini" for the Gini impurity and "entropy" for the information gain. Defaults to "gini".
- **max_features** (*int, float or {"auto", "sqrt", "log2"}*) – The number of features to consider when looking for the best split:
  - If int, then consider max_features features at each split.
  - If float, then max_features is a fraction and int(max_features * n_features) features are considered at each split.
  - If "auto", then max_features=sqrt(n_features).
  - If "sqrt", then max_features=sqrt(n_features).
  - If "log2", then max_features=log2(n_features).
  - If None, then max_features = n_features.

  The search for a split does not stop until at least one valid partition of the node samples is found, even if it requires to effectively inspect more than max_features features. Defaults to "auto".
- **max_depth** (*int*) – The maximum depth of the tree. Defaults to 6.
- **min_samples_split** (*int or float*) – The minimum number of samples required to split an internal node:
  - If int, then consider min_samples_split as the minimum number.
  - If float, then min_samples_split is a fraction and ceil(min_samples_split * n_samples) are the minimum number of samples for each split.

  Defaults to 2.
- **min_weight_fraction_leaf** (*float*) – The minimum weighted fraction of the sum total of weights (of all the input samples) required to be at a leaf node. Defaults to 0.0.
- **random_seed** (*int*) – Seed for the random number generator. Defaults to 0.

**Attributes**

| | |
|---|---|
| **hyper-parameter_ranges** | { "criterion": ["gini", "entropy"], "max_features": ["auto", "sqrt", "log2"], "max_depth": Integer(4, 10),} |
| **model_family** | ModelFamily.DECISION_TREE |
| **modifies_features** | True |
| **modifies_target** | False |
| **name** | Decision Tree Classifier |
| **supported_problem_types** | [ ProblemTypes.BINARY, ProblemTypes.MULTICLASS, ProblemTypes.TIME_SERIES_BINARY, ProblemTypes.TIME_SERIES_MULTICLASS,] |
| **training_only** | False |

**Methods**

| | |
|---|---|
| *clone* | Constructs a new component with the same parameters and random state. |
| *default_parameters* | Returns the default parameters for this component. |
| *describe* | Describe a component and its parameters. |
| *feature_importance* | Returns importance associated with each feature. |
| *fit* | Fits estimator to data. |
| *get_prediction_intervals* | Find the prediction intervals using the fitted regressor. |
| *load* | Loads component at file path. |
| *needs_fitting* | Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances. |
| *parameters* | Returns the parameters which were used to initialize the component. |
| *predict* | Make predictions using selected features. |
| *predict_proba* | Make probability estimates for labels. |
| *save* | Saves component at file path. |
| *update_parameters* | Updates the parameter dictionary of the component. |

**clone**(*self*)

Constructs a new component with the same parameters and random state.

**Returns** A new instance of this component with identical parameters and random state.

**default_parameters**(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

**Returns** Default parameters for this component.

**Return type** dict

**describe**(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

**Parameters**

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description as dictionary in the format {"name": name, "parameters": parameters}

**Returns** Returns dictionary if return_dict is True, else None.

**Return type** None or dict

**property feature_importance**(*self*) → pandas.Series

Returns importance associated with each feature.

**Returns** Importance associated with each feature.

**Return type** np.ndarray

**Raises** **MethodPropertyNotFoundError** – If estimator does not have a feature_importance method or a component_obj that implements feature_importance.

**fit**(*self*, *X: pandas.DataFrame*, *y: Optional[pandas.Series] = None*)

Fits estimator to data.

**Parameters**

- **X** (*pd.DataFrame*) – The input training data of shape [n_samples, n_features].
- **y** (*pd.Series, optional*) – The target training data of length [n_samples].

**Returns** self

**get_prediction_intervals**(*self, X: pandas.DataFrame, y: Optional[pandas.Series] = None, coverage: List[float] = None, predictions: pandas.Series = None*) → Dict[str, pandas.Series]

Find the prediction intervals using the fitted regressor.

This function takes the predictions of the fitted estimator and calculates the rolling standard deviation across all predictions using a window size of 5. The lower and upper predictions are determined by taking the percent point (quantile) function of the lower tail probability at each bound multiplied by the rolling standard deviation.

**Parameters**

- **X** (*pd.DataFrame*) – Data of shape [n_samples, n_features].
- **y** (*pd.Series*) – Target data. Ignored.
- **coverage** (*list[float]*) – A list of floats between the values 0 and 1 that the upper and lower bounds of the prediction interval should be calculated for.
- **predictions** (*pd.Series*) – Optional list of predictions to use. If None, will generate predictions using *X*.

**Returns** Prediction intervals, keys are in the format {coverage}_lower or {coverage}_upper.

**Return type** dict

**Raises** **MethodPropertyNotFoundError** – If the estimator does not support Time Series Regression as a problem type.

**static load**(*file_path*)

Loads component at file path.

**Parameters** **file_path** (*str*) – Location to load file.

**Returns** ComponentBase object

**needs_fitting**(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need to be fit or whose fit methods do nothing.

**Returns** True.

**property parameters**(*self*)

Returns the parameters which were used to initialize the component.

**predict**(*self, X: pandas.DataFrame*) → pandas.Series

Make predictions using selected features.

**Parameters** **X** (*pd.DataFrame*) – Data of shape [n_samples, n_features].

**Returns** Predicted values.

**Return type** pd.Series

**Raises** `MethodPropertyNotFoundError` – If estimator does not have a predict method or a component_obj that implements predict.

`predict_proba`(*self*, *X: pandas.DataFrame*) → pandas.Series

Make probability estimates for labels.

**Parameters** `X` (*pd.DataFrame*) – Features.

**Returns** Probability estimates.

**Return type** pd.Series

**Raises** `MethodPropertyNotFoundError` – If estimator does not have a predict_proba method or a component_obj that implements predict_proba.

`save`(*self*, *file_path*, *pickle_protocol=cloudpickle.DEFAULT_PROTOCOL*)

Saves component at file path.

**Parameters**

- `file_path` (*str*) – Location to save file.
- `pickle_protocol` (*int*) – The pickle data stream format.

`update_parameters`(*self*, *update_dict*, *reset_fit=True*)

Updates the parameter dictionary of the component.

**Parameters**

- `update_dict` (*dict*) – A dict of parameters to update.
- `reset_fit` (*bool, optional*) – If True, will set *_is_fitted* to False.

**class** `evalml.pipelines.components.estimators.classifiers.`**`ElasticNetClassifier`**(*penalty='elasticnet', C=1.0, l1_ratio=0.15, multi_class='auto', solver='saga', n_jobs=- 1, random_seed=0, **kwargs*)

Elastic Net Classifier. Uses Logistic Regression with elasticnet penalty as the base estimator.

**Parameters**

- `penalty` (*{"l1", "l2", "elasticnet", "none"}*) – The norm used in penalization. Defaults to "elasticnet".
- `C` (*float*) – Inverse of regularization strength. Must be a positive float. Defaults to 1.0.
- `l1_ratio` (*float*) – The mixing parameter, with 0 <= l1_ratio <= 1. Only used if penalty='elasticnet'. Setting l1_ratio=0 is equivalent to using penalty='l2', while setting l1_ratio=1 is equivalent to using penalty='l1'. For 0 < l1_ratio <1, the penalty is a combination of L1 and L2. Defaults to 0.15.
- `multi_class` (*{"auto", "ovr", "multinomial"}*) – If the option chosen is "ovr", then a binary problem is fit for each label. For "multinomial" the loss minimised is the multinomial loss fit across the entire probability distribution, even when the data is binary. "multinomial" is unavailable when solver="liblinear". "auto" selects "ovr" if the data is binary, or if solver="liblinear", and otherwise selects "multinomial". Defaults to "auto".

- **solver** (*{"newton-cg", "lbfgs", "liblinear", "sag", "saga"}*) – Algorithm to use in the optimization problem. For small datasets, "liblinear" is a good choice, whereas "sag" and "saga" are faster for large ones. For multiclass problems, only "newton-cg", "sag", "saga" and "lbfgs" handle multinomial loss; "liblinear" is limited to one-versus-rest schemes.
  - "newton-cg", "lbfgs", "sag" and "saga" handle L2 or no penalty
  - "liblinear" and "saga" also handle L1 penalty
  - "saga" also supports "elasticnet" penalty
  - "liblinear" does not support setting penalty='none'

  Defaults to "saga".
- **n_jobs** (*int*) – Number of parallel threads used to run xgboost. Note that creating thread contention will significantly slow down the algorithm. Defaults to -1.
- **random_seed** (*int*) – Seed for the random number generator. Defaults to 0.

**Attributes**

| | |
|---|---|
| **hyper-parameter_ranges** | { "C": Real(0.01, 10), "l1_ratio": Real(0, 1)} |
| **model_family** | ModelFamily.LINEAR_MODEL |
| **modifies_features** | True |
| **modifies_target** | False |
| **name** | Elastic Net Classifier |
| **supported_problem_types** | [ ProblemTypes.BINARY, ProblemTypes.MULTICLASS, ProblemTypes.TIME_SERIES_BINARY, ProblemTypes.TIME_SERIES_MULTICLASS,] |
| **training_only** | False |

**Methods**

| | |
|---|---|
| *clone* | Constructs a new component with the same parameters and random state. |
| *default_parameters* | Returns the default parameters for this component. |
| *describe* | Describe a component and its parameters. |
| *feature_importance* | Feature importance for fitted ElasticNet classifier. |
| *fit* | Fits ElasticNet classifier component to data. |
| *get_prediction_intervals* | Find the prediction intervals using the fitted regressor. |
| *load* | Loads component at file path. |
| *needs_fitting* | Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances. |
| *parameters* | Returns the parameters which were used to initialize the component. |
| *predict* | Make predictions using selected features. |
| *predict_proba* | Make probability estimates for labels. |
| *save* | Saves component at file path. |
| *update_parameters* | Updates the parameter dictionary of the component. |

**clone**(*self*)

Constructs a new component with the same parameters and random state.

**Returns** A new instance of this component with identical parameters and random state.

**default_parameters**(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

**Returns** Default parameters for this component.

**Return type** dict

**describe**(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

**Parameters**

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description as dictionary in the format {"name": name, "parameters": parameters}

**Returns** Returns dictionary if return_dict is True, else None.

**Return type** None or dict

**property feature_importance**(*self*)

Feature importance for fitted ElasticNet classifier.

**fit**(*self*, *X*, *y*)

Fits ElasticNet classifier component to data.

**Parameters**

- **X** (*pd.DataFrame*) – The input training data of shape [n_samples, n_features].
- **y** (*pd.Series*) – The target training data of length [n_samples].

**Returns** self

**get_prediction_intervals**(*self*, *X: pandas.DataFrame*, *y: Optional[pandas.Series] = None*, *coverage: List[float] = None*, *predictions: pandas.Series = None*) → Dict[str, pandas.Series]

Find the prediction intervals using the fitted regressor.

This function takes the predictions of the fitted estimator and calculates the rolling standard deviation across all predictions using a window size of 5. The lower and upper predictions are determined by taking the percent point (quantile) function of the lower tail probability at each bound multiplied by the rolling standard deviation.

**Parameters**

- **X** (*pd.DataFrame*) – Data of shape [n_samples, n_features].
- **y** (*pd.Series*) – Target data. Ignored.
- **coverage** (*list[float]*) – A list of floats between the values 0 and 1 that the upper and lower bounds of the prediction interval should be calculated for.
- **predictions** (*pd.Series*) – Optional list of predictions to use. If None, will generate predictions using *X*.

**Returns** Prediction intervals, keys are in the format {coverage}_lower or {coverage}_upper.

**Return type** dict

**Raises** `MethodPropertyNotFoundError` – If the estimator does not support Time Series Regression as a problem type.

`static load`(*file_path*)

Loads component at file path.

**Parameters** `file_path` (*str*) – Location to load file.

**Returns** ComponentBase object

`needs_fitting`(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need to be fit or whose fit methods do nothing.

**Returns** True.

`property parameters`(*self*)

Returns the parameters which were used to initialize the component.

`predict`(*self*, *X: pandas.DataFrame*) → pandas.Series

Make predictions using selected features.

**Parameters** `X` (*pd.DataFrame*) – Data of shape [n_samples, n_features].

**Returns** Predicted values.

**Return type** pd.Series

**Raises** `MethodPropertyNotFoundError` – If estimator does not have a predict method or a component_obj that implements predict.

`predict_proba`(*self*, *X: pandas.DataFrame*) → pandas.Series

Make probability estimates for labels.

**Parameters** `X` (*pd.DataFrame*) – Features.

**Returns** Probability estimates.

**Return type** pd.Series

**Raises** `MethodPropertyNotFoundError` – If estimator does not have a predict_proba method or a component_obj that implements predict_proba.

`save`(*self*, *file_path*, *pickle_protocol=cloudpickle.DEFAULT_PROTOCOL*)

Saves component at file path.

**Parameters**

- `file_path` (*str*) – Location to save file.
- `pickle_protocol` (*int*) – The pickle data stream format.

`update_parameters`(*self*, *update_dict*, *reset_fit=True*)

Updates the parameter dictionary of the component.

**Parameters**

- `update_dict` (*dict*) – A dict of parameters to update.
- `reset_fit` (*bool, optional*) – If True, will set *_is_fitted* to False.

**class** evalml.pipelines.components.estimators.classifiers.**ExtraTreesClassifier**(*n_estimators=100*, *max_features='auto'*, *max_depth=6*, *min_samples_split=2*, *min_weight_fraction_leaf=0.0*, *n_jobs=- 1*, *random_seed=0*, ***kwargs*)

Extra Trees Classifier.

**Parameters**

- **n_estimators** (*float*) – The number of trees in the forest. Defaults to 100.
- **max_features** (*int, float or {"auto", "sqrt", "log2"}*) – The number of features to consider when looking for the best split:
  - If int, then consider max_features features at each split.
  - If float, then max_features is a fraction and int(max_features * n_features) features are considered at each split.
  - If "auto", then max_features=sqrt(n_features).
  - If "sqrt", then max_features=sqrt(n_features).
  - If "log2", then max_features=log2(n_features).
  - If None, then max_features = n_features.

  The search for a split does not stop until at least one valid partition of the node samples is found, even if it requires to effectively inspect more than max_features features. Defaults to "auto".
- **max_depth** (*int*) – The maximum depth of the tree. Defaults to 6.
- **min_samples_split** (*int or float*) – The minimum number of samples required to split an internal node:
  - If int, then consider min_samples_split as the minimum number.
  - If float, then min_samples_split is a fraction and ceil(min_samples_split * n_samples) are the minimum number of samples for each split.
- **2.** (*Defaults to*) –
- **min_weight_fraction_leaf** (*float*) – The minimum weighted fraction of the sum total of weights (of all the input samples) required to be at a leaf node. Defaults to 0.0.
- **n_jobs** (*int or None*) – Number of jobs to run in parallel. -1 uses all processes. Defaults to -1.
- **random_seed** (*int*) – Seed for the random number generator. Defaults to 0.

**Attributes**

| | |
|---|---|
| **hyper-parameter_ranges** | { "n_estimators": Integer(10, 1000), "max_features": ["auto", "sqrt", "log2"], "max_depth": Integer(4, 10),} |
| **model_family** | ModelFamily.EXTRA_TREES |
| **modifies_features** | True |
| **modifies_target** | False |
| **name** | Extra Trees Classifier |
| **supported_problem_types** | [ ProblemTypes.BINARY, ProblemTypes.MULTICLASS, ProblemTypes.TIME_SERIES_BINARY, ProblemTypes.TIME_SERIES_MULTICLASS,] |
| **training_only** | False |

**Methods**

| | |
|---|---|
| *clone* | Constructs a new component with the same parameters and random state. |
| *default_parameters* | Returns the default parameters for this component. |
| *describe* | Describe a component and its parameters. |
| *feature_importance* | Returns importance associated with each feature. |
| *fit* | Fits estimator to data. |
| *get_prediction_intervals* | Find the prediction intervals using the fitted regressor. |
| *load* | Loads component at file path. |
| *needs_fitting* | Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances. |
| *parameters* | Returns the parameters which were used to initialize the component. |
| *predict* | Make predictions using selected features. |
| *predict_proba* | Make probability estimates for labels. |
| *save* | Saves component at file path. |
| *update_parameters* | Updates the parameter dictionary of the component. |

**clone**(*self*)

Constructs a new component with the same parameters and random state.

**Returns** A new instance of this component with identical parameters and random state.

**default_parameters**(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

**Returns** Default parameters for this component.

**Return type** dict

**describe**(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

**Parameters**

- **print_name** (*bool, optional*) – whether to print name of component

- **return_dict** (*bool, optional*) – whether to return description as dictionary in the format {"name": name, "parameters": parameters}

**Returns** Returns dictionary if return_dict is True, else None.

**Return type** None or dict

**property feature_importance**(*self*) → pandas.Series

Returns importance associated with each feature.

**Returns** Importance associated with each feature.

**Return type** np.ndarray

**Raises** **MethodPropertyNotFoundError** – If estimator does not have a feature_importance method or a component_obj that implements feature_importance.

**fit**(*self, X: pandas.DataFrame, y: Optional[pandas.Series] = None*)

Fits estimator to data.

**Parameters**

- **X** (*pd.DataFrame*) – The input training data of shape [n_samples, n_features].
- **y** (*pd.Series, optional*) – The target training data of length [n_samples].

**Returns** self

**get_prediction_intervals**(*self, X: pandas.DataFrame, y: Optional[pandas.Series] = None, coverage: List[float] = None, predictions: pandas.Series = None*) → Dict[str, pandas.Series]

Find the prediction intervals using the fitted regressor.

This function takes the predictions of the fitted estimator and calculates the rolling standard deviation across all predictions using a window size of 5. The lower and upper predictions are determined by taking the percent point (quantile) function of the lower tail probability at each bound multiplied by the rolling standard deviation.

**Parameters**

- **X** (*pd.DataFrame*) – Data of shape [n_samples, n_features].
- **y** (*pd.Series*) – Target data. Ignored.
- **coverage** (*list[float]*) – A list of floats between the values 0 and 1 that the upper and lower bounds of the prediction interval should be calculated for.
- **predictions** (*pd.Series*) – Optional list of predictions to use. If None, will generate predictions using *X*.

**Returns** Prediction intervals, keys are in the format {coverage}_lower or {coverage}_upper.

**Return type** dict

**Raises** **MethodPropertyNotFoundError** – If the estimator does not support Time Series Regression as a problem type.

**static load**(*file_path*)

Loads component at file path.

**Parameters** **file_path** (*str*) – Location to load file.

**Returns** ComponentBase object

**needs_fitting**(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need to be fit or whose fit methods do nothing.

**Returns** True.

**property parameters**(*self*)

Returns the parameters which were used to initialize the component.

**predict**(*self*, *X: pandas.DataFrame*) → pandas.Series

Make predictions using selected features.

**Parameters** **X** (*pd.DataFrame*) – Data of shape [n_samples, n_features].

**Returns** Predicted values.

**Return type** pd.Series

**Raises** **MethodPropertyNotFoundError** – If estimator does not have a predict method or a component_obj that implements predict.

**predict_proba**(*self*, *X: pandas.DataFrame*) → pandas.Series

Make probability estimates for labels.

**Parameters** **X** (*pd.DataFrame*) – Features.

**Returns** Probability estimates.

**Return type** pd.Series

**Raises** **MethodPropertyNotFoundError** – If estimator does not have a predict_proba method or a component_obj that implements predict_proba.

**save**(*self*, *file_path*, *pickle_protocol=cloudpickle.DEFAULT_PROTOCOL*)

Saves component at file path.

**Parameters**

- **file_path** (*str*) – Location to save file.
- **pickle_protocol** (*int*) – The pickle data stream format.

**update_parameters**(*self*, *update_dict*, *reset_fit=True*)

Updates the parameter dictionary of the component.

**Parameters**

- **update_dict** (*dict*) – A dict of parameters to update.
- **reset_fit** (*bool, optional*) – If True, will set *_is_fitted* to False.

**class** evalml.pipelines.components.estimators.classifiers.**KNeighborsClassifier**(*n_neighbors=5*, *weights='uniform'*, *algorithm='auto'*, *leaf_size=30*, *p=2*, *random_seed=0*, ***kwargs*)

K-Nearest Neighbors Classifier.

**Parameters**

- **n_neighbors** (*int*) – Number of neighbors to use by default. Defaults to 5.
- **weights** (*{'uniform', 'distance'} or callable*) – Weight function used in prediction. Can be:
  - 'uniform' : uniform weights. All points in each neighborhood are weighted equally.
  - 'distance' : weight points by the inverse of their distance. in this case, closer neighbors of a query point will have a greater influence than neighbors which are further away.
  - [callable] : a user-defined function which accepts an array of distances, and returns an array of the same shape containing the weights.

  Defaults to "uniform".
- **algorithm** (*{'auto', 'ball_tree', 'kd_tree', 'brute'}*) – Algorithm used to compute the nearest neighbors:
  - 'ball_tree' will use BallTree
  - 'kd_tree' will use KDTree
  - 'brute' will use a brute-force search.

  'auto' will attempt to decide the most appropriate algorithm based on the values passed to fit method. Defaults to "auto". Note: fitting on sparse input will override the setting of this parameter, using brute force.
- **leaf_size** (*int*) – Leaf size passed to BallTree or KDTree. This can affect the speed of the construction and query, as well as the memory required to store the tree. The optimal value depends on the nature of the problem. Defaults to 30.
- **p** (*int*) – Power parameter for the Minkowski metric. When p = 1, this is equivalent to using manhattan_distance (l1), and euclidean_distance (l2) for p = 2. For arbitrary p, minkowski_distance (l_p) is used. Defaults to 2.
- **random_seed** (*int*) – Seed for the random number generator. Defaults to 0.

**Attributes**

| | |
|---|---|
| **hyper-parameter_ranges** | { "n_neighbors": Integer(2, 12), "weights": ["uniform", "distance"], "algorithm": ["auto", "ball_tree", "kd_tree", "brute"], "leaf_size": Integer(10, 30), "p": Integer(1, 5),} |
| **model_family** | ModelFamily.K_NEIGHBORS |
| **modifies_features** | True |
| **modifies_target** | False |
| **name** | KNN Classifier |
| **supported_problem_types** | [ ProblemTypes.BINARY, ProblemTypes.MULTICLASS, ProblemTypes.TIME_SERIES_BINARY, ProblemTypes.TIME_SERIES_MULTICLASS,] |
| **training_only** | False |

**Methods**

| | |
|---|---|
| *clone* | Constructs a new component with the same parameters and random state. |
| *default_parameters* | Returns the default parameters for this component. |
| *describe* | Describe a component and its parameters. |
| *feature_importance* | Returns array of 0's matching the input number of features as feature_importance is not defined for KNN classifiers. |
| *fit* | Fits estimator to data. |
| *get_prediction_intervals* | Find the prediction intervals using the fitted regressor. |
| *load* | Loads component at file path. |
| *needs_fitting* | Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances. |
| *parameters* | Returns the parameters which were used to initialize the component. |
| *predict* | Make predictions using selected features. |
| *predict_proba* | Make probability estimates for labels. |
| *save* | Saves component at file path. |
| *update_parameters* | Updates the parameter dictionary of the component. |

**clone**(*self*)

Constructs a new component with the same parameters and random state.

**Returns** A new instance of this component with identical parameters and random state.

**default_parameters**(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

**Returns** Default parameters for this component.

**Return type** dict

**describe**(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

**Parameters**

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description as dictionary in the format {"name": name, "parameters": parameters}

**Returns** Returns dictionary if return_dict is True, else None.

**Return type** None or dict

**property feature_importance**(*self*)

Returns array of 0's matching the input number of features as feature_importance is not defined for KNN classifiers.

**fit**(*self*, *X: pandas.DataFrame*, *y: Optional[pandas.Series] = None*)

Fits estimator to data.

**Parameters**

- **X** (*pd.DataFrame*) – The input training data of shape [n_samples, n_features].

- **y** (*pd.Series, optional*) – The target training data of length [n_samples].

**Returns** self

**get_prediction_intervals**(*self, X: pandas.DataFrame, y: Optional[pandas.Series] = None, coverage: List[float] = None, predictions: pandas.Series = None*) → Dict[str, pandas.Series]

Find the prediction intervals using the fitted regressor.

This function takes the predictions of the fitted estimator and calculates the rolling standard deviation across all predictions using a window size of 5. The lower and upper predictions are determined by taking the percent point (quantile) function of the lower tail probability at each bound multiplied by the rolling standard deviation.

**Parameters**

- **X** (*pd.DataFrame*) – Data of shape [n_samples, n_features].
- **y** (*pd.Series*) – Target data. Ignored.
- **coverage** (*list[float]*) – A list of floats between the values 0 and 1 that the upper and lower bounds of the prediction interval should be calculated for.
- **predictions** (*pd.Series*) – Optional list of predictions to use. If None, will generate predictions using *X*.

**Returns** Prediction intervals, keys are in the format {coverage}_lower or {coverage}_upper.

**Return type** dict

**Raises** **MethodPropertyNotFoundError** – If the estimator does not support Time Series Regression as a problem type.

**static load**(*file_path*)

Loads component at file path.

**Parameters** **file_path** (*str*) – Location to load file.

**Returns** ComponentBase object

**needs_fitting**(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need to be fit or whose fit methods do nothing.

**Returns** True.

**property parameters**(*self*)

Returns the parameters which were used to initialize the component.

**predict**(*self, X: pandas.DataFrame*) → pandas.Series

Make predictions using selected features.

**Parameters** **X** (*pd.DataFrame*) – Data of shape [n_samples, n_features].

**Returns** Predicted values.

**Return type** pd.Series

**Raises** **MethodPropertyNotFoundError** – If estimator does not have a predict method or a component_obj that implements predict.

**predict_proba**(*self*, *X: pandas.DataFrame*) → pandas.Series

Make probability estimates for labels.

**Parameters** **X** (*pd.DataFrame*) – Features.

**Returns** Probability estimates.

**Return type** pd.Series

**Raises** **MethodPropertyNotFoundError** – If estimator does not have a predict_proba method or a component_obj that implements predict_proba.

**save**(*self*, *file_path*, *pickle_protocol=cloudpickle.DEFAULT_PROTOCOL*)

Saves component at file path.

**Parameters**

- **file_path** (*str*) – Location to save file.
- **pickle_protocol** (*int*) – The pickle data stream format.

**update_parameters**(*self*, *update_dict*, *reset_fit=True*)

Updates the parameter dictionary of the component.

**Parameters**

- **update_dict** (*dict*) – A dict of parameters to update.
- **reset_fit** (*bool, optional*) – If True, will set *_is_fitted* to False.

**class** evalml.pipelines.components.estimators.classifiers.**LightGBMClassifier**(*boosting_type='gbdt', learning_rate=0.1, n_estimators=100, max_depth=0, num_leaves=31, min_child_samples=20, bagging_fraction=0.9, bagging_freq=0, n_jobs=- 1, random_seed=0, **kwargs*)

LightGBM Classifier.

**Parameters**

- **boosting_type** (*string*) – Type of boosting to use. Defaults to "gbdt". - 'gbdt' uses traditional Gradient Boosting Decision Tree - "dart", uses Dropouts meet Multiple Additive Regression Trees - "goss", uses Gradient-based One-Side Sampling - "rf", uses Random Forest
- **learning_rate** (*float*) – Boosting learning rate. Defaults to 0.1.
- **n_estimators** (*int*) – Number of boosted trees to fit. Defaults to 100.
- **max_depth** (*int*) – Maximum tree depth for base learners, <=0 means no limit. Defaults to 0.
- **num_leaves** (*int*) – Maximum tree leaves for base learners. Defaults to 31.
- **min_child_samples** (*int*) – Minimum number of data needed in a child (leaf). Defaults to 20.

- **bagging_fraction** (*float*) – LightGBM will randomly select a subset of features on each iteration (tree) without resampling if this is smaller than 1.0. For example, if set to 0.8, LightGBM will select 80% of features before training each tree. This can be used to speed up training and deal with overfitting. Defaults to 0.9.
- **bagging_freq** (*int*) – Frequency for bagging. 0 means bagging is disabled. k means perform bagging at every k iteration. Every k-th iteration, LightGBM will randomly select bagging_fraction * 100 % of the data to use for the next k iterations. Defaults to 0.
- **n_jobs** (*int or None*) – Number of threads to run in parallel. -1 uses all threads. Defaults to -1.
- **random_seed** (*int*) – Seed for the random number generator. Defaults to 0.

**Attributes**

| | |
|---|---|
| **hyper-parameter_ranges** | { "learning_rate": Real(0.000001, 1), "boosting_type": ["gbdt", "dart", "goss", "rf"], "n_estimators": Integer(10, 100), "max_depth": Integer(0, 10), "num_leaves": Integer(2, 100), "min_child_samples": Integer(1, 100), "bagging_fraction": Real(0.000001, 1), "bagging_freq": Integer(0, 1),} |
| **model_family** | ModelFamily.LIGHTGBM |
| **modifies_features** | True |
| **modifies_target** | False |
| **name** | LightGBM Classifier |
| **SEED_MAX** | SEED_BOUNDS.max_bound |
| **SEED_MIN** | 0 |
| **supported_problem_types** | [ ProblemTypes.BINARY, ProblemTypes.MULTICLASS, ProblemTypes.TIME_SERIES_BINARY, ProblemTypes.TIME_SERIES_MULTICLASS,] |
| **training_only** | False |

**Methods**

| | |
|---|---|
| *clone* | Constructs a new component with the same parameters and random state. |
| *default_parameters* | Returns the default parameters for this component. |
| *describe* | Describe a component and its parameters. |
| *feature_importance* | Returns importance associated with each feature. |
| *fit* | Fits LightGBM classifier component to data. |
| *get_prediction_intervals* | Find the prediction intervals using the fitted regressor. |
| *load* | Loads component at file path. |
| *needs_fitting* | Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances. |
| *parameters* | Returns the parameters which were used to initialize the component. |
| *predict* | Make predictions using the fitted LightGBM classifier. |
| *predict_proba* | Make prediction probabilities using the fitted LightGBM classifier. |
| *save* | Saves component at file path. |
| *update_parameters* | Updates the parameter dictionary of the component. |

**clone**(*self*)

Constructs a new component with the same parameters and random state.

**Returns** A new instance of this component with identical parameters and random state.

**default_parameters**(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

**Returns** Default parameters for this component.

**Return type** dict

**describe**(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

**Parameters**

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description as dictionary in the format {"name": name, "parameters": parameters}

**Returns** Returns dictionary if return_dict is True, else None.

**Return type** None or dict

**property feature_importance**(*self*) → pandas.Series

Returns importance associated with each feature.

**Returns** Importance associated with each feature.

**Return type** np.ndarray

**Raises** **MethodPropertyNotFoundError** – If estimator does not have a feature_importance method or a component_obj that implements feature_importance.

**fit**(*self*, *X*, *y=None*)

Fits LightGBM classifier component to data.

**Parameters**

- **X** (*pd.DataFrame*) – The input training data of shape [n_samples, n_features].
- **y** (*pd.Series*) – The target training data of length [n_samples].

**Returns** self

**get_prediction_intervals**(*self*, *X: pandas.DataFrame*, *y: Optional[pandas.Series] = None*, *coverage: List[float] = None*, *predictions: pandas.Series = None*) → Dict[str, pandas.Series]

Find the prediction intervals using the fitted regressor.

This function takes the predictions of the fitted estimator and calculates the rolling standard deviation across all predictions using a window size of 5. The lower and upper predictions are determined by taking the percent point (quantile) function of the lower tail probability at each bound multiplied by the rolling standard deviation.

**Parameters**

- **X** (*pd.DataFrame*) – Data of shape [n_samples, n_features].
- **y** (*pd.Series*) – Target data. Ignored.

- **coverage** (*list[float]*) – A list of floats between the values 0 and 1 that the upper and lower bounds of the prediction interval should be calculated for.
- **predictions** (*pd.Series*) – Optional list of predictions to use. If None, will generate predictions using *X*.

**Returns** Prediction intervals, keys are in the format {coverage}_lower or {coverage}_upper.

**Return type** dict

**Raises** **MethodPropertyNotFoundError** – If the estimator does not support Time Series Regression as a problem type.

**static load**(*file_path*)

Loads component at file path.

**Parameters** **file_path** (*str*) – Location to load file.

**Returns** ComponentBase object

**needs_fitting**(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need to be fit or whose fit methods do nothing.

**Returns** True.

**property parameters**(*self*)

Returns the parameters which were used to initialize the component.

**predict**(*self*, *X*)

Make predictions using the fitted LightGBM classifier.

**Parameters** **X** (*pd.DataFrame*) – Data of shape [n_samples, n_features].

**Returns** Predicted values.

**Return type** pd.DataFrame

**predict_proba**(*self*, *X*)

Make prediction probabilities using the fitted LightGBM classifier.

**Parameters** **X** (*pd.DataFrame*) – Data of shape [n_samples, n_features].

**Returns** Predicted probability values.

**Return type** pd.DataFrame

**save**(*self*, *file_path*, *pickle_protocol=cloudpickle.DEFAULT_PROTOCOL*)

Saves component at file path.

**Parameters**

- **file_path** (*str*) – Location to save file.
- **pickle_protocol** (*int*) – The pickle data stream format.

**update_parameters**(*self*, *update_dict*, *reset_fit=True*)

Updates the parameter dictionary of the component.

**Parameters**

- **update_dict** (*dict*) – A dict of parameters to update.
- **reset_fit** (*bool, optional*) – If True, will set *_is_fitted* to False.

**class** evalml.pipelines.components.estimators.classifiers.**LogisticRegressionClassifier**(*penalty='l2', C=1.0, multi_class='auto', solver='lbfgs', n_jobs=-1, random_seed=0, **kwargs*)

Logistic Regression Classifier.

**Parameters**

- **penalty** (*{"l1", "l2", "elasticnet", "none"}*) – The norm used in penalization. Defaults to "l2".
- **C** (*float*) – Inverse of regularization strength. Must be a positive float. Defaults to 1.0.
- **multi_class** (*{"auto", "ovr", "multinomial"}*) – If the option chosen is "ovr", then a binary problem is fit for each label. For "multinomial" the loss minimised is the multinomial loss fit across the entire probability distribution, even when the data is binary. "multinomial" is unavailable when solver="liblinear". "auto" selects "ovr" if the data is binary, or if solver="liblinear", and otherwise selects "multinomial". Defaults to "auto".
- **solver** (*{"newton-cg", "lbfgs", "liblinear", "sag", "saga"}*) – Algorithm to use in the optimization problem. For small datasets, "liblinear" is a good choice, whereas "sag" and "saga" are faster for large ones. For multiclass problems, only "newton-cg", "sag", "saga" and "lbfgs" handle multinomial loss; "liblinear" is limited to one-versus-rest schemes.
  - "newton-cg", "lbfgs", "sag" and "saga" handle L2 or no penalty
  - "liblinear" and "saga" also handle L1 penalty
  - "saga" also supports "elasticnet" penalty
  - "liblinear" does not support setting penalty='none'

  Defaults to "lbfgs".
- **n_jobs** (*int*) – Number of parallel threads used to run xgboost. Note that creating thread contention will significantly slow down the algorithm. Defaults to -1.
- **random_seed** (*int*) – Seed for the random number generator. Defaults to 0.

**Attributes**

| | |
|---|---|
| **hyperparameter_ranges** | { "penalty": ["l2"], "C": Real(0.01, 10),} |
| **model_family** | ModelFamily.LINEAR_MODEL |
| **modifies_features** | True |
| **modifies_target** | False |
| **name** | Logistic Regression Classifier |
| **supported_problem_types** | [ ProblemTypes.BINARY, ProblemTypes.MULTICLASS, ProblemTypes.TIME_SERIES_BINARY, ProblemTypes.TIME_SERIES_MULTICLASS,] |
| **training_only** | False |

**Methods**

| | |
|---|---|
| *clone* | Constructs a new component with the same parameters and random state. |
| *default_parameters* | Returns the default parameters for this component. |
| *describe* | Describe a component and its parameters. |
| *feature_importance* | Feature importance for fitted logistic regression classifier. |
| *fit* | Fits estimator to data. |
| *get_prediction_intervals* | Find the prediction intervals using the fitted regressor. |
| *load* | Loads component at file path. |
| *needs_fitting* | Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances. |
| *parameters* | Returns the parameters which were used to initialize the component. |
| *predict* | Make predictions using selected features. |
| *predict_proba* | Make probability estimates for labels. |
| *save* | Saves component at file path. |
| *update_parameters* | Updates the parameter dictionary of the component. |

**clone**(*self*)

Constructs a new component with the same parameters and random state.

**Returns** A new instance of this component with identical parameters and random state.

**default_parameters**(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

**Returns** Default parameters for this component.

**Return type** dict

**describe**(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

**Parameters**

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description as dictionary in the format {"name": name, "parameters": parameters}

**Returns** Returns dictionary if return_dict is True, else None.

**Return type** None or dict

**property feature_importance**(*self*)

Feature importance for fitted logistic regression classifier.

**fit**(*self*, *X: pandas.DataFrame*, *y: Optional[pandas.Series] = None*)

Fits estimator to data.

**Parameters**

- **X** (*pd.DataFrame*) – The input training data of shape [n_samples, n_features].

- **y** (*pd.Series, optional*) – The target training data of length [n_samples].

**Returns** self

**get_prediction_intervals**(*self, X: pandas.DataFrame, y: Optional[pandas.Series] = None, coverage: List[float] = None, predictions: pandas.Series = None*) → Dict[str, pandas.Series]

Find the prediction intervals using the fitted regressor.

This function takes the predictions of the fitted estimator and calculates the rolling standard deviation across all predictions using a window size of 5. The lower and upper predictions are determined by taking the percent point (quantile) function of the lower tail probability at each bound multiplied by the rolling standard deviation.

**Parameters**

- **X** (*pd.DataFrame*) – Data of shape [n_samples, n_features].
- **y** (*pd.Series*) – Target data. Ignored.
- **coverage** (*list[float]*) – A list of floats between the values 0 and 1 that the upper and lower bounds of the prediction interval should be calculated for.
- **predictions** (*pd.Series*) – Optional list of predictions to use. If None, will generate predictions using *X*.

**Returns** Prediction intervals, keys are in the format {coverage}_lower or {coverage}_upper.

**Return type** dict

**Raises** **MethodPropertyNotFoundError** – If the estimator does not support Time Series Regression as a problem type.

**static load**(*file_path*)

Loads component at file path.

**Parameters** **file_path** (*str*) – Location to load file.

**Returns** ComponentBase object

**needs_fitting**(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need to be fit or whose fit methods do nothing.

**Returns** True.

**property parameters**(*self*)

Returns the parameters which were used to initialize the component.

**predict**(*self, X: pandas.DataFrame*) → pandas.Series

Make predictions using selected features.

**Parameters** **X** (*pd.DataFrame*) – Data of shape [n_samples, n_features].

**Returns** Predicted values.

**Return type** pd.Series

**Raises** **MethodPropertyNotFoundError** – If estimator does not have a predict method or a component_obj that implements predict.

**predict_proba**(*self*, *X: pandas.DataFrame*) → pandas.Series

Make probability estimates for labels.

**Parameters** **X** (*pd.DataFrame*) – Features.

**Returns** Probability estimates.

**Return type** pd.Series

**Raises** **MethodPropertyNotFoundError** – If estimator does not have a predict_proba method or a component_obj that implements predict_proba.

**save**(*self*, *file_path*, *pickle_protocol=cloudpickle.DEFAULT_PROTOCOL*)

Saves component at file path.

**Parameters**

- **file_path** (*str*) – Location to save file.
- **pickle_protocol** (*int*) – The pickle data stream format.

**update_parameters**(*self*, *update_dict*, *reset_fit=True*)

Updates the parameter dictionary of the component.

**Parameters**

- **update_dict** (*dict*) – A dict of parameters to update.
- **reset_fit** (*bool, optional*) – If True, will set *_is_fitted* to False.

**class** evalml.pipelines.components.estimators.classifiers.**RandomForestClassifier**(*n_estimators=100*, *max_depth=6*, *n_jobs=- 1*, *random_seed=0*, ***kwargs*)

Random Forest Classifier.

**Parameters**

- **n_estimators** (*float*) – The number of trees in the forest. Defaults to 100.
- **max_depth** (*int*) – Maximum tree depth for base learners. Defaults to 6.
- **n_jobs** (*int or None*) – Number of jobs to run in parallel. -1 uses all processes. Defaults to -1.
- **random_seed** (*int*) – Seed for the random number generator. Defaults to 0.

**Attributes**

| | |
|---|---|
| **hyper-parameter_ranges** | { "n_estimators": Integer(10, 1000), "max_depth": Integer(1, 10),} |
| **model_family** | ModelFamily.RANDOM_FOREST |
| **modifies_features** | True |
| **modifies_target** | False |
| **name** | Random Forest Classifier |
| **supported_problem_types** | [ ProblemTypes.BINARY, ProblemTypes.MULTICLASS, ProblemTypes.TIME_SERIES_BINARY, ProblemTypes.TIME_SERIES_MULTICLASS,] |
| **training_only** | False |

**Methods**

| | |
|---|---|
| *clone* | Constructs a new component with the same parameters and random state. |
| *default_parameters* | Returns the default parameters for this component. |
| *describe* | Describe a component and its parameters. |
| *feature_importance* | Returns importance associated with each feature. |
| *fit* | Fits estimator to data. |
| *get_prediction_intervals* | Find the prediction intervals using the fitted regressor. |
| *load* | Loads component at file path. |
| *needs_fitting* | Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances. |
| *parameters* | Returns the parameters which were used to initialize the component. |
| *predict* | Make predictions using selected features. |
| *predict_proba* | Make probability estimates for labels. |
| *save* | Saves component at file path. |
| *update_parameters* | Updates the parameter dictionary of the component. |

**clone**(*self*)

Constructs a new component with the same parameters and random state.

**Returns** A new instance of this component with identical parameters and random state.

**default_parameters**(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

**Returns** Default parameters for this component.

**Return type** dict

**describe**(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

**Parameters**

- **print_name** (*bool, optional*) – whether to print name of component

- **return_dict** (*bool, optional*) – whether to return description as dictionary in the format {"name": name, "parameters": parameters}

**Returns** Returns dictionary if return_dict is True, else None.

**Return type** None or dict

**property feature_importance**(*self*) → pandas.Series

Returns importance associated with each feature.

**Returns** Importance associated with each feature.

**Return type** np.ndarray

**Raises** **MethodPropertyNotFoundError** – If estimator does not have a feature_importance method or a component_obj that implements feature_importance.

**fit**(*self, X: pandas.DataFrame, y: Optional[pandas.Series] = None*)

Fits estimator to data.

**Parameters**

- **X** (*pd.DataFrame*) – The input training data of shape [n_samples, n_features].
- **y** (*pd.Series, optional*) – The target training data of length [n_samples].

**Returns** self

**get_prediction_intervals**(*self, X: pandas.DataFrame, y: Optional[pandas.Series] = None, coverage: List[float] = None, predictions: pandas.Series = None*) → Dict[str, pandas.Series]

Find the prediction intervals using the fitted regressor.

This function takes the predictions of the fitted estimator and calculates the rolling standard deviation across all predictions using a window size of 5. The lower and upper predictions are determined by taking the percent point (quantile) function of the lower tail probability at each bound multiplied by the rolling standard deviation.

**Parameters**

- **X** (*pd.DataFrame*) – Data of shape [n_samples, n_features].
- **y** (*pd.Series*) – Target data. Ignored.
- **coverage** (*list[float]*) – A list of floats between the values 0 and 1 that the upper and lower bounds of the prediction interval should be calculated for.
- **predictions** (*pd.Series*) – Optional list of predictions to use. If None, will generate predictions using *X*.

**Returns** Prediction intervals, keys are in the format {coverage}_lower or {coverage}_upper.

**Return type** dict

**Raises** **MethodPropertyNotFoundError** – If the estimator does not support Time Series Regression as a problem type.

**static load**(*file_path*)

Loads component at file path.

**Parameters** **file_path** (*str*) – Location to load file.

**Returns** ComponentBase object

**needs_fitting**(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need to be fit or whose fit methods do nothing.

**Returns** True.

**property parameters**(*self*)

Returns the parameters which were used to initialize the component.

**predict**(*self*, *X: pandas.DataFrame*) → pandas.Series

Make predictions using selected features.

**Parameters** **X** (*pd.DataFrame*) – Data of shape [n_samples, n_features].

**Returns** Predicted values.

**Return type** pd.Series

**Raises** **MethodPropertyNotFoundError** – If estimator does not have a predict method or a component_obj that implements predict.

**predict_proba**(*self*, *X: pandas.DataFrame*) → pandas.Series

Make probability estimates for labels.

**Parameters** **X** (*pd.DataFrame*) – Features.

**Returns** Probability estimates.

**Return type** pd.Series

**Raises** **MethodPropertyNotFoundError** – If estimator does not have a predict_proba method or a component_obj that implements predict_proba.

**save**(*self*, *file_path*, *pickle_protocol=cloudpickle.DEFAULT_PROTOCOL*)

Saves component at file path.

**Parameters**

- **file_path** (*str*) – Location to save file.
- **pickle_protocol** (*int*) – The pickle data stream format.

**update_parameters**(*self*, *update_dict*, *reset_fit=True*)

Updates the parameter dictionary of the component.

**Parameters**

- **update_dict** (*dict*) – A dict of parameters to update.
- **reset_fit** (*bool, optional*) – If True, will set *_is_fitted* to False.

**class** evalml.pipelines.components.estimators.classifiers.**SVMClassifier**(*C=1.0*, *kernel='rbf'*, *gamma='auto'*, *probability=True*, *random_seed=0*, ***kwargs*)

Support Vector Machine Classifier.

**Parameters**

- **C** (*float*) – The regularization parameter. The strength of the regularization is inversely proportional to C. Must be strictly positive. The penalty is a squared l2 penalty. Defaults to 1.0.
- **kernel** (*{"poly", "rbf", "sigmoid"}*) – Specifies the kernel type to be used in the algorithm. Defaults to "rbf".
- **gamma** (*{"scale", "auto"} or float*) – Kernel coefficient for "rbf", "poly" and "sigmoid". Defaults to "auto". - If gamma='scale' is passed then it uses 1 / (n_features * X.var()) as value of gamma - If "auto" (default), uses 1 / n_features
- **probability** (*boolean*) – Whether to enable probability estimates. Defaults to True.
- **random_seed** (*int*) – Seed for the random number generator. Defaults to 0.

**Attributes**

| | |
|---|---|
| **hyper-parameter_ranges** | { "C": Real(0, 10), "kernel": ["poly", "rbf", "sigmoid"], "gamma": ["scale", "auto"],} |
| **model_family** | ModelFamily.SVM |
| **modifies_features** | True |
| **modifies_target** | False |
| **name** | SVM Classifier |
| **supported_problem_types** | [ ProblemTypes.BINARY, ProblemTypes.MULTICLASS, ProblemTypes.TIME_SERIES_BINARY, ProblemTypes.TIME_SERIES_MULTICLASS,] |
| **training_only** | False |

**Methods**

| | |
|---|---|
| *clone* | Constructs a new component with the same parameters and random state. |
| *default_parameters* | Returns the default parameters for this component. |
| *describe* | Describe a component and its parameters. |
| *feature_importance* | Feature importance only works with linear kernels. |
| *fit* | Fits estimator to data. |
| *get_prediction_intervals* | Find the prediction intervals using the fitted regressor. |
| *load* | Loads component at file path. |
| *needs_fitting* | Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances. |
| *parameters* | Returns the parameters which were used to initialize the component. |
| *predict* | Make predictions using selected features. |
| *predict_proba* | Make probability estimates for labels. |
| *save* | Saves component at file path. |
| *update_parameters* | Updates the parameter dictionary of the component. |

**clone**(*self*)

Constructs a new component with the same parameters and random state.

**Returns** A new instance of this component with identical parameters and random state.

**default_parameters**(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

**Returns** Default parameters for this component.

**Return type** dict

**describe**(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

**Parameters**

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description as dictionary in the format {"name": name, "parameters": parameters}

**Returns** Returns dictionary if return_dict is True, else None.

**Return type** None or dict

**property feature_importance**(*self*)

Feature importance only works with linear kernels.

If the kernel isn't linear, we return a numpy array of zeros.

**Returns** Feature importance of fitted SVM classifier or a numpy array of zeroes if the kernel is not linear.

**fit**(*self*, *X: pandas.DataFrame*, *y: Optional[pandas.Series] = None*)

Fits estimator to data.

**Parameters**

- **X** (*pd.DataFrame*) – The input training data of shape [n_samples, n_features].
- **y** (*pd.Series, optional*) – The target training data of length [n_samples].

**Returns** self

**get_prediction_intervals**(*self*, *X: pandas.DataFrame*, *y: Optional[pandas.Series] = None*, *coverage: List[float] = None*, *predictions: pandas.Series = None*) → Dict[str, pandas.Series]

Find the prediction intervals using the fitted regressor.

This function takes the predictions of the fitted estimator and calculates the rolling standard deviation across all predictions using a window size of 5. The lower and upper predictions are determined by taking the percent point (quantile) function of the lower tail probability at each bound multiplied by the rolling standard deviation.

**Parameters**

- **X** (*pd.DataFrame*) – Data of shape [n_samples, n_features].
- **y** (*pd.Series*) – Target data. Ignored.
- **coverage** (*list[float]*) – A list of floats between the values 0 and 1 that the upper and lower bounds of the prediction interval should be calculated for.
- **predictions** (*pd.Series*) – Optional list of predictions to use. If None, will generate predictions using *X*.

**Returns** Prediction intervals, keys are in the format {coverage}_lower or {coverage}_upper.

**Return type** dict

**Raises** `MethodPropertyNotFoundError` – If the estimator does not support Time Series Regression as a problem type.

`static load`(*file_path*)

Loads component at file path.

**Parameters** `file_path` (*str*) – Location to load file.

**Returns** ComponentBase object

`needs_fitting`(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need to be fit or whose fit methods do nothing.

**Returns** True.

`property parameters`(*self*)

Returns the parameters which were used to initialize the component.

`predict`(*self*, *X: pandas.DataFrame*) → pandas.Series

Make predictions using selected features.

**Parameters** `X` (*pd.DataFrame*) – Data of shape [n_samples, n_features].

**Returns** Predicted values.

**Return type** pd.Series

**Raises** `MethodPropertyNotFoundError` – If estimator does not have a predict method or a component_obj that implements predict.

`predict_proba`(*self*, *X: pandas.DataFrame*) → pandas.Series

Make probability estimates for labels.

**Parameters** `X` (*pd.DataFrame*) – Features.

**Returns** Probability estimates.

**Return type** pd.Series

**Raises** `MethodPropertyNotFoundError` – If estimator does not have a predict_proba method or a component_obj that implements predict_proba.

`save`(*self*, *file_path*, *pickle_protocol=cloudpickle.DEFAULT_PROTOCOL*)

Saves component at file path.

**Parameters**

- `file_path` (*str*) – Location to save file.
- `pickle_protocol` (*int*) – The pickle data stream format.

`update_parameters`(*self*, *update_dict*, *reset_fit=True*)

Updates the parameter dictionary of the component.

**Parameters**

- `update_dict` (*dict*) – A dict of parameters to update.
- `reset_fit` (*bool, optional*) – If True, will set *_is_fitted* to False.

**class** evalml.pipelines.components.estimators.classifiers.**VowpalWabbitBinaryClassifier**(*loss_function='logistic learning_rate=0.5, decay_learning_rate=1.0 power_t=0.5, passes=1, random_seed=0, **kwargs*)

Vowpal Wabbit Binary Classifier.

**Parameters**

- **loss_function** (*str*) – Specifies the loss function to use. One of {"squared", "classic", "hinge", "logistic", "quantile"}. Defaults to "logistic".
- **learning_rate** (*float*) – Boosting learning rate. Defaults to 0.5.
- **decay_learning_rate** (*float*) – Decay factor for learning_rate. Defaults to 1.0.
- **power_t** (*float*) – Power on learning rate decay. Defaults to 0.5.
- **passes** (*int*) – Number of training passes. Defaults to 1.
- **random_seed** (*int*) – Seed for the random number generator. Defaults to 0.

**Attributes**

| | |
|---|---|
| **hyperparameter_ranges** | None |
| **model_family** | ModelFamily.VOWPAL_WABBIT |
| **modifies_features** | True |
| **modifies_target** | False |
| **name** | Vowpal Wabbit Binary Classifier |
| **supported_problem_types** | [ ProblemTypes.BINARY, ProblemTypes.TIME_SERIES_BINARY,] |
| **training_only** | False |

**Methods**

| | |
|---|---|
| *clone* | Constructs a new component with the same parameters and random state. |
| *default_parameters* | Returns the default parameters for this component. |
| *describe* | Describe a component and its parameters. |
| *feature_importance* | Feature importance for Vowpal Wabbit classifiers. This is not implemented. |
| *fit* | Fits estimator to data. |
| *get_prediction_intervals* | Find the prediction intervals using the fitted regressor. |
| *load* | Loads component at file path. |
| *needs_fitting* | Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances. |
| *parameters* | Returns the parameters which were used to initialize the component. |
| *predict* | Make predictions using selected features. |
| *predict_proba* | Make probability estimates for labels. |
| *save* | Saves component at file path. |
| *update_parameters* | Updates the parameter dictionary of the component. |

**clone**(*self*)

Constructs a new component with the same parameters and random state.

**Returns** A new instance of this component with identical parameters and random state.

**default_parameters**(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

**Returns** Default parameters for this component.

**Return type** dict

**describe**(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

**Parameters**

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description as dictionary in the format {"name": name, "parameters": parameters}

**Returns** Returns dictionary if return_dict is True, else None.

**Return type** None or dict

**property feature_importance**(*self*)

Feature importance for Vowpal Wabbit classifiers. This is not implemented.

**fit**(*self*, *X: pandas.DataFrame*, *y: Optional[pandas.Series] = None*)

Fits estimator to data.

**Parameters**

- **X** (*pd.DataFrame*) – The input training data of shape [n_samples, n_features].
- **y** (*pd.Series, optional*) – The target training data of length [n_samples].

**Returns** self

**get_prediction_intervals**(*self*, *X: pandas.DataFrame*, *y: Optional[pandas.Series] = None*, *coverage: List[float] = None*, *predictions: pandas.Series = None*) → Dict[str, pandas.Series]

Find the prediction intervals using the fitted regressor.

This function takes the predictions of the fitted estimator and calculates the rolling standard deviation across all predictions using a window size of 5. The lower and upper predictions are determined by taking the percent point (quantile) function of the lower tail probability at each bound multiplied by the rolling standard deviation.

**Parameters**

- **X** (*pd.DataFrame*) – Data of shape [n_samples, n_features].
- **y** (*pd.Series*) – Target data. Ignored.
- **coverage** (*list[float]*) – A list of floats between the values 0 and 1 that the upper and lower bounds of the prediction interval should be calculated for.
- **predictions** (*pd.Series*) – Optional list of predictions to use. If None, will generate predictions using *X*.

**Returns** Prediction intervals, keys are in the format {coverage}_lower or {coverage}_upper.

**Return type** dict

**Raises** **MethodPropertyNotFoundError** – If the estimator does not support Time Series Regression as a problem type.

**static load**(*file_path*)

Loads component at file path.

**Parameters** **file_path** (*str*) – Location to load file.

**Returns** ComponentBase object

**needs_fitting**(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need to be fit or whose fit methods do nothing.

**Returns** True.

**property parameters**(*self*)

Returns the parameters which were used to initialize the component.

**predict**(*self*, *X: pandas.DataFrame*) → pandas.Series

Make predictions using selected features.

**Parameters** **X** (*pd.DataFrame*) – Data of shape [n_samples, n_features].

**Returns** Predicted values.

**Return type** pd.Series

**Raises** **MethodPropertyNotFoundError** – If estimator does not have a predict method or a component_obj that implements predict.

**predict_proba**(*self*, *X: pandas.DataFrame*) → pandas.Series

Make probability estimates for labels.

**Parameters** **X** (*pd.DataFrame*) – Features.

**Returns** Probability estimates.

**Return type** pd.Series

**Raises** **MethodPropertyNotFoundError** – If estimator does not have a predict_proba method or a component_obj that implements predict_proba.

**save**(*self*, *file_path*, *pickle_protocol=cloudpickle.DEFAULT_PROTOCOL*)

Saves component at file path.

**Parameters**

- **file_path** (*str*) – Location to save file.
- **pickle_protocol** (*int*) – The pickle data stream format.

**update_parameters**(*self*, *update_dict*, *reset_fit=True*)

Updates the parameter dictionary of the component.

**Parameters**

- **update_dict** (*dict*) – A dict of parameters to update.
- **reset_fit** (*bool, optional*) – If True, will set *_is_fitted* to False.

**class** evalml.pipelines.components.estimators.classifiers.**VowpalWabbitMulticlassClassifier**(*loss_function='lo learn- ing_rate=0.5, de- cay_learning_rat power_t=0.5, passes=1, ran- dom_seed=0, **kwargs*)

Vowpal Wabbit Multiclass Classifier.

**Parameters**

- **loss_function** (*str*) – Specifies the loss function to use. One of {"squared", "classic", "hinge", "logistic", "quantile"}. Defaults to "logistic".
- **learning_rate** (*float*) – Boosting learning rate. Defaults to 0.5.
- **decay_learning_rate** (*float*) – Decay factor for learning_rate. Defaults to 1.0.
- **power_t** (*float*) – Power on learning rate decay. Defaults to 0.5.
- **random_seed** (*int*) – Seed for the random number generator. Defaults to 0.

**Attributes**

| | |
|---|---|
| **hyper-parameter_ranges** | None |
| **model_family** | ModelFamily.VOWPAL_WABBIT |
| **modifies_features** | True |
| **modifies_target** | False |
| **name** | Vowpal Wabbit Multiclass Classifier |
| **supported_problem_types** | [ ProblemTypes.MULTICLASS, ProblemTypes.TIME_SERIES_MULTICLASS,] |
| **training_only** | False |

**Methods**

| | |
|---|---|
| *clone* | Constructs a new component with the same parameters and random state. |
| *default_parameters* | Returns the default parameters for this component. |
| *describe* | Describe a component and its parameters. |
| *feature_importance* | Feature importance for Vowpal Wabbit classifiers. This is not implemented. |
| *fit* | Fits estimator to data. |
| *get_prediction_intervals* | Find the prediction intervals using the fitted regressor. |
| *load* | Loads component at file path. |
| *needs_fitting* | Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances. |
| *parameters* | Returns the parameters which were used to initialize the component. |
| *predict* | Make predictions using selected features. |
| *predict_proba* | Make probability estimates for labels. |
| *save* | Saves component at file path. |
| *update_parameters* | Updates the parameter dictionary of the component. |

**clone**(*self*)

Constructs a new component with the same parameters and random state.

**Returns** A new instance of this component with identical parameters and random state.

**default_parameters**(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

**Returns** Default parameters for this component.

**Return type** dict

**describe**(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

**Parameters**

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description as dictionary in the format {"name": name, "parameters": parameters}

**Returns** Returns dictionary if return_dict is True, else None.

**Return type** None or dict

**property feature_importance**(*self*)

Feature importance for Vowpal Wabbit classifiers. This is not implemented.

**fit**(*self*, *X: pandas.DataFrame*, *y: Optional[pandas.Series] = None*)

Fits estimator to data.

**Parameters**

- **X** (*pd.DataFrame*) – The input training data of shape [n_samples, n_features].
- **y** (*pd.Series, optional*) – The target training data of length [n_samples].

**Returns** self

**get_prediction_intervals**(*self*, *X: pandas.DataFrame*, *y: Optional[pandas.Series] = None*, *coverage: List[float] = None*, *predictions: pandas.Series = None*) → Dict[str, pandas.Series]

Find the prediction intervals using the fitted regressor.

This function takes the predictions of the fitted estimator and calculates the rolling standard deviation across all predictions using a window size of 5. The lower and upper predictions are determined by taking the percent point (quantile) function of the lower tail probability at each bound multiplied by the rolling standard deviation.

**Parameters**

- **X** (*pd.DataFrame*) – Data of shape [n_samples, n_features].
- **y** (*pd.Series*) – Target data. Ignored.
- **coverage** (*list[float]*) – A list of floats between the values 0 and 1 that the upper and lower bounds of the prediction interval should be calculated for.
- **predictions** (*pd.Series*) – Optional list of predictions to use. If None, will generate predictions using *X*.

**Returns** Prediction intervals, keys are in the format {coverage}_lower or {coverage}_upper.

**Return type** dict

**Raises** **MethodPropertyNotFoundError** – If the estimator does not support Time Series Regression as a problem type.

**static load**(*file_path*)

Loads component at file path.

**Parameters** **file_path** (*str*) – Location to load file.

**Returns** ComponentBase object

**needs_fitting**(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need to be fit or whose fit methods do nothing.

**Returns** True.

**property parameters**(*self*)

Returns the parameters which were used to initialize the component.

**predict**(*self*, *X: pandas.DataFrame*) → pandas.Series

Make predictions using selected features.

**Parameters** **X** (*pd.DataFrame*) – Data of shape [n_samples, n_features].

**Returns** Predicted values.

**Return type** pd.Series

**Raises** **MethodPropertyNotFoundError** – If estimator does not have a predict method or a component_obj that implements predict.

**predict_proba**(*self*, *X: pandas.DataFrame*) → pandas.Series

Make probability estimates for labels.

**Parameters** **X** (*pd.DataFrame*) – Features.

**Returns** Probability estimates.

**Return type** pd.Series

**Raises** **MethodPropertyNotFoundError** – If estimator does not have a predict_proba method or a component_obj that implements predict_proba.

**save**(*self*, *file_path*, *pickle_protocol=cloudpickle.DEFAULT_PROTOCOL*)

Saves component at file path.

**Parameters**

- **file_path** (*str*) – Location to save file.
- **pickle_protocol** (*int*) – The pickle data stream format.

**update_parameters**(*self*, *update_dict*, *reset_fit=True*)

Updates the parameter dictionary of the component.

**Parameters**

- **update_dict** (*dict*) – A dict of parameters to update.
- **reset_fit** (*bool, optional*) – If True, will set *_is_fitted* to False.

**class** evalml.pipelines.components.estimators.classifiers.**XGBoostClassifier**(*eta=0.1*, *max_depth=6*, *min_child_weight=1*, *n_estimators=100*, *random_seed=0*, *eval_metric='logloss'*, *n_jobs=12*, ***kwargs*)

XGBoost Classifier.

**Parameters**

- **eta** (*float*) – Boosting learning rate. Defaults to 0.1.
- **max_depth** (*int*) – Maximum tree depth for base learners. Defaults to 6.
- **min_child_weight** (*float*) – Minimum sum of instance weight (hessian) needed in a child. Defaults to 1.0

- **n_estimators** (*int*) – Number of gradient boosted trees. Equivalent to number of boosting rounds. Defaults to 100.
- **random_seed** (*int*) – Seed for the random number generator. Defaults to 0.
- **n_jobs** (*int*) – Number of parallel threads used to run xgboost. Note that creating thread contention will significantly slow down the algorithm. Defaults to 12.

**Attributes**

| | |
|---|---|
| **hyper-parameter_ranges** | { "eta": Real(0.000001, 1), "max_depth": Integer(1, 10), "min_child_weight": Real(1, 10), "n_estimators": Integer(1, 1000),} |
| **model_family** | ModelFamily.XGBOOST |
| **modifies_features** | True |
| **modifies_target** | False |
| **name** | XGBoost Classifier |
| **SEED_MAX** | None |
| **SEED_MIN** | None |
| **supported_problem_types** | [ ProblemTypes.BINARY, ProblemTypes.MULTICLASS, ProblemTypes.TIME_SERIES_BINARY, ProblemTypes.TIME_SERIES_MULTICLASS,] |
| **training_only** | False |

**Methods**

| | |
|---|---|
| *clone* | Constructs a new component with the same parameters and random state. |
| *default_parameters* | Returns the default parameters for this component. |
| *describe* | Describe a component and its parameters. |
| *feature_importance* | Feature importance of fitted XGBoost classifier. |
| *fit* | Fits XGBoost classifier component to data. |
| *get_prediction_intervals* | Find the prediction intervals using the fitted regressor. |
| *load* | Loads component at file path. |
| *needs_fitting* | Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances. |
| *parameters* | Returns the parameters which were used to initialize the component. |
| *predict* | Make predictions using the fitted XGBoost classifier. |
| *predict_proba* | Make predictions using the fitted CatBoost classifier. |
| *save* | Saves component at file path. |
| *update_parameters* | Updates the parameter dictionary of the component. |

**clone**(*self*)

Constructs a new component with the same parameters and random state.

**Returns** A new instance of this component with identical parameters and random state.

**default_parameters**(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

**Returns** Default parameters for this component.

**Return type** dict

**describe**(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

**Parameters**

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description as dictionary in the format {"name": name, "parameters": parameters}

**Returns** Returns dictionary if return_dict is True, else None.

**Return type** None or dict

**property feature_importance**(*self*)

Feature importance of fitted XGBoost classifier.

**fit**(*self*, *X*, *y=None*)

Fits XGBoost classifier component to data.

**Parameters**

- **X** (*pd.DataFrame*) – The input training data of shape [n_samples, n_features].
- **y** (*pd.Series*) – The target training data of length [n_samples].

**Returns** self

**get_prediction_intervals**(*self*, *X: pandas.DataFrame*, *y: Optional[pandas.Series] = None*, *coverage: List[float] = None*, *predictions: pandas.Series = None*) → Dict[str, pandas.Series]

Find the prediction intervals using the fitted regressor.

This function takes the predictions of the fitted estimator and calculates the rolling standard deviation across all predictions using a window size of 5. The lower and upper predictions are determined by taking the percent point (quantile) function of the lower tail probability at each bound multiplied by the rolling standard deviation.

**Parameters**

- **X** (*pd.DataFrame*) – Data of shape [n_samples, n_features].
- **y** (*pd.Series*) – Target data. Ignored.
- **coverage** (*list[float]*) – A list of floats between the values 0 and 1 that the upper and lower bounds of the prediction interval should be calculated for.
- **predictions** (*pd.Series*) – Optional list of predictions to use. If None, will generate predictions using *X*.

**Returns** Prediction intervals, keys are in the format {coverage}_lower or {coverage}_upper.

**Return type** dict

**Raises** **MethodPropertyNotFoundError** – If the estimator does not support Time Series Regression as a problem type.

**static load**(*file_path*)

Loads component at file path.

**Parameters** **file_path** (*str*) – Location to load file.

**Returns** ComponentBase object

**needs_fitting**(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need to be fit or whose fit methods do nothing.

**Returns** True.

**property parameters**(*self*)

Returns the parameters which were used to initialize the component.

**predict**(*self*, *X*)

Make predictions using the fitted XGBoost classifier.

**Parameters** **X** (*pd.DataFrame*) – Data of shape [n_samples, n_features].

**Returns** Predicted values.

**Return type** pd.DataFrame

**predict_proba**(*self*, *X*)

Make predictions using the fitted CatBoost classifier.

**Parameters** **X** (*pd.DataFrame*) – Data of shape [n_samples, n_features].

**Returns** Predicted values.

**Return type** pd.DataFrame

**save**(*self*, *file_path*, *pickle_protocol=cloudpickle.DEFAULT_PROTOCOL*)

Saves component at file path.

**Parameters**

- **file_path** (*str*) – Location to save file.
- **pickle_protocol** (*int*) – The pickle data stream format.

**update_parameters**(*self*, *update_dict*, *reset_fit=True*)

Updates the parameter dictionary of the component.

**Parameters**

- **update_dict** (*dict*) – A dict of parameters to update.
- **reset_fit** (*bool, optional*) – If True, will set *_is_fitted* to False.

### regressors

Regression model components.

### Submodules

### arima_regressor

Autoregressive Integrated Moving Average Model. The three parameters (p, d, q) are the AR order, the degree of differencing, and the MA order. More information here: https://www.statsmodels.org/devel/generated/statsmodels.tsa.arima.model.ARIMA.html.

### Module Contents

### Classes Summary

| | |
|---|---|
| *ARIMARegressor* | Autoregressive Integrated Moving Average Model. The three parameters (p, d, q) are the AR order, the degree of differencing, and the MA order. More information here: https://www.statsmodels.org/devel/generated/statsmodels.tsa.arima.model.ARIMA.html. |

### Contents

**class** evalml.pipelines.components.estimators.regressors.arima_regressor.**ARIMARegressor**(*time_index: Optional[Hashable] = None, trend: Optional[str] = None, start_p: int = 2, d: int = 0, start_q: int = 2, max_p: int = 5, max_d: int = 2, max_q: int = 5, seasonal: bool = True, sp: int = 1, n_jobs: int = -1, random_seed: Union[int, float] = 0, maxiter: int = 10, use_covariates: bool*

Autoregressive Integrated Moving Average Model. The three parameters (p, d, q) are the AR order, the degree of differencing, and the MA order. More information here: https://www.statsmodels.org/devel/generated/statsmodels.tsa.arima.model.ARIMA.html.

Currently ARIMARegressor isn't supported via conda install. It's recommended that it be installed via PyPI.

**Parameters**

- **time_index** (*str*) – Specifies the name of the column in X that provides the datetime objects. Defaults to None.
- **trend** (*str*) – Controls the deterministic trend. Options are ['n', 'c', 't', 'ct'] where 'c' is a constant term, 't' indicates a linear trend, and 'ct' is both. Can also be an iterable when defining a polynomial, such as [1, 1, 0, 1].
- **start_p** (*int*) – Minimum Autoregressive order. Defaults to 2.
- **d** (*int*) – Minimum Differencing degree. Defaults to 0.
- **start_q** (*int*) – Minimum Moving Average order. Defaults to 2.
- **max_p** (*int*) – Maximum Autoregressive order. Defaults to 5.
- **max_d** (*int*) – Maximum Differencing degree. Defaults to 2.
- **max_q** (*int*) – Maximum Moving Average order. Defaults to 5.
- **seasonal** (*boolean*) – Whether to fit a seasonal model to ARIMA. Defaults to True.
- **sp** (*int or str*) – Period for seasonal differencing, specifically the number of periods in each season. If "detect", this model will automatically detect this parameter (given the time series is a standard frequency) and will fall back to 1 (no seasonality) if it cannot be detected. Defaults to 1.
- **n_jobs** (*int or None*) – Non-negative integer describing level of parallelism used for pipelines. Defaults to -1.
- **random_seed** (*int*) – Seed for the random number generator. Defaults to 0.

**Attributes**

| | |
|---|---|
| **hyper-parameter_ranges** | { "start_p": Integer(1, 3), "d": Integer(0, 2), "start_q": Integer(1, 3), "max_p": Integer(3, 10), "max_d": Integer(2, 5), "max_q": Integer(3, 10), "seasonal": [True, False],} |
| **max_cols** | 7 |
| **max_rows** | 1000 |
| **model_family** | ModelFamily.ARIMA |
| **modifies_features** | True |
| **modifies_target** | False |
| **name** | ARIMA Regressor |
| **supported_problem_types** | [ProblemTypes.TIME_SERIES_REGRESSION] |
| **training_only** | False |

**Methods**

| | |
|---|---|
| *clone* | Constructs a new component with the same parameters and random state. |
| *default_parameters* | Returns the default parameters for this component. |
| *describe* | Describe a component and its parameters. |
| *feature_importance* | Returns array of 0's with a length of 1 as feature_importance is not defined for ARIMA regressor. |
| *fit* | Fits ARIMA regressor to data. |
| *get_prediction_intervals* | Find the prediction intervals using the fitted ARIMARegressor. |
| *load* | Loads component at file path. |
| *needs_fitting* | Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances. |
| *parameters* | Returns the parameters which were used to initialize the component. |
| *predict* | Make predictions using fitted ARIMA regressor. |
| *predict_proba* | Make probability estimates for labels. |
| *save* | Saves component at file path. |
| *update_parameters* | Updates the parameter dictionary of the component. |

**clone**(*self*)

Constructs a new component with the same parameters and random state.

**Returns** A new instance of this component with identical parameters and random state.

**default_parameters**(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

**Returns** Default parameters for this component.

**Return type** dict

**describe**(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

**Parameters**

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description as dictionary in the format {"name": name, "parameters": parameters}

**Returns** Returns dictionary if return_dict is True, else None.

**Return type** None or dict

**property feature_importance**(*self*) → numpy.ndarray

Returns array of 0's with a length of 1 as feature_importance is not defined for ARIMA regressor.

**fit**(*self*, *X: pandas.DataFrame*, *y: Optional[pandas.Series] = None*)

Fits ARIMA regressor to data.

**Parameters**

- **X** (*pd.DataFrame*) – The input training data of shape [n_samples, n_features].
- **y** (*pd.Series*) – The target training data of length [n_samples].

**Returns** self

**Raises** `ValueError` – If y was not passed in.

`get_prediction_intervals`(*self, X: pandas.DataFrame, y: pandas.Series = None, coverage: List[float] = None, predictions: pandas.Series = None*) → Dict[str, pandas.Series]

Find the prediction intervals using the fitted ARIMARegressor.

**Parameters**

- `X` (*pd.DataFrame*) – Data of shape [n_samples, n_features].
- `y` (*pd.Series*) – Target data. Optional.
- `coverage` (*list[float]*) – A list of floats between the values 0 and 1 that the upper and lower bounds of the prediction interval should be calculated for.
- `predictions` (*pd.Series*) – Not used for ARIMA regressor.

**Returns** Prediction intervals, keys are in the format {coverage}_lower or {coverage}_upper.

**Return type** dict

`static load`(*file_path*)

Loads component at file path.

**Parameters** `file_path` (*str*) – Location to load file.

**Returns** ComponentBase object

`needs_fitting`(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need to be fit or whose fit methods do nothing.

**Returns** True.

`property parameters`(*self*)

Returns the parameters which were used to initialize the component.

`predict`(*self, X: pandas.DataFrame, y: Optional[pandas.Series] = None*) → pandas.Series

Make predictions using fitted ARIMA regressor.

**Parameters**

- `X` (*pd.DataFrame*) – Data of shape [n_samples, n_features].
- `y` (*pd.Series*) – Target data.

**Returns** Predicted values.

**Return type** pd.Series

**Raises** `ValueError` – If X was passed to *fit* but not passed in *predict*.

`predict_proba`(*self, X: pandas.DataFrame*) → pandas.Series

Make probability estimates for labels.

**Parameters** `X` (*pd.DataFrame*) – Features.

**Returns** Probability estimates.

**Return type** pd.Series

**Raises** `MethodPropertyNotFoundError` – If estimator does not have a predict_proba method or a component_obj that implements predict_proba.

**save**(*self*, *file_path*, *pickle_protocol=cloudpickle.DEFAULT_PROTOCOL*)

Saves component at file path.

**Parameters**

- **file_path** (*str*) – Location to save file.
- **pickle_protocol** (*int*) – The pickle data stream format.

**update_parameters**(*self*, *update_dict*, *reset_fit=True*)

Updates the parameter dictionary of the component.

**Parameters**

- **update_dict** (*dict*) – A dict of parameters to update.
- **reset_fit** (*bool, optional*) – If True, will set *_is_fitted* to False.

## baseline_regressor

Baseline regressor that uses a simple strategy to make predictions. This is useful as a simple baseline regressor to compare with other regressors.

### Module Contents

### Classes Summary

| | |
|---|---|
| *BaselineRegressor* | Baseline regressor that uses a simple strategy to make predictions. This is useful as a simple baseline regressor to compare with other regressors. |

### Contents

**class** evalml.pipelines.components.estimators.regressors.baseline_regressor.**BaselineRegressor**(*strategy='mea ran-dom_seed=0, **kwargs*)

Baseline regressor that uses a simple strategy to make predictions. This is useful as a simple baseline regressor to compare with other regressors.

**Parameters**

- **strategy** (*str*) – Method used to predict. Valid options are "mean", "median". Defaults to "mean".
- **random_seed** (*int*) – Seed for the random number generator. Defaults to 0.

**Attributes**

| | |
|---|---|
| **hyper-parameter_ranges** | {} |
| **model_family** | ModelFamily.BASELINE |
| **modifies_features** | True |
| **modifies_target** | False |
| **name** | Baseline Regressor |
| **supported_problem_types** | [ ProblemTypes.REGRESSION, ProblemTypes.TIME_SERIES_REGRESSION,] |
| **training_only** | False |

**Methods**

| | |
|---|---|
| *clone* | Constructs a new component with the same parameters and random state. |
| *default_parameters* | Returns the default parameters for this component. |
| *describe* | Describe a component and its parameters. |
| *feature_importance* | Returns importance associated with each feature. Since baseline regressors do not use input features to calculate predictions, returns an array of zeroes. |
| *fit* | Fits baseline regression component to data. |
| *get_prediction_intervals* | Find the prediction intervals using the fitted regressor. |
| *load* | Loads component at file path. |
| *needs_fitting* | Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances. |
| *parameters* | Returns the parameters which were used to initialize the component. |
| *predict* | Make predictions using the baseline regression strategy. |
| *predict_proba* | Make probability estimates for labels. |
| *save* | Saves component at file path. |
| *update_parameters* | Updates the parameter dictionary of the component. |

**clone**(*self*)

Constructs a new component with the same parameters and random state.

**Returns** A new instance of this component with identical parameters and random state.

**default_parameters**(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

**Returns** Default parameters for this component.

**Return type** dict

**describe**(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

**Parameters**

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description as dictionary in the format {"name": name, "parameters": parameters}

**Returns** Returns dictionary if return_dict is True, else None.

**Return type** None or dict

**property feature_importance**(*self*)

Returns importance associated with each feature. Since baseline regressors do not use input features to calculate predictions, returns an array of zeroes.

**Returns** An array of zeroes.

**Return type** np.ndarray (float)

**fit**(*self*, *X*, *y=None*)

Fits baseline regression component to data.

**Parameters**

- **X** (*pd.DataFrame*) – The input training data of shape [n_samples, n_features].
- **y** (*pd.Series*) – The target training data of length [n_samples].

**Returns** self

**Raises** **ValueError** – If input y is None.

**get_prediction_intervals**(*self*, *X: pandas.DataFrame*, *y: Optional[pandas.Series] = None*, *coverage: List[float] = None*, *predictions: pandas.Series = None*) → Dict[str, pandas.Series]

Find the prediction intervals using the fitted regressor.

This function takes the predictions of the fitted estimator and calculates the rolling standard deviation across all predictions using a window size of 5. The lower and upper predictions are determined by taking the percent point (quantile) function of the lower tail probability at each bound multiplied by the rolling standard deviation.

**Parameters**

- **X** (*pd.DataFrame*) – Data of shape [n_samples, n_features].
- **y** (*pd.Series*) – Target data. Ignored.
- **coverage** (*list[float]*) – A list of floats between the values 0 and 1 that the upper and lower bounds of the prediction interval should be calculated for.
- **predictions** (*pd.Series*) – Optional list of predictions to use. If None, will generate predictions using *X*.

**Returns** Prediction intervals, keys are in the format {coverage}_lower or {coverage}_upper.

**Return type** dict

**Raises** **MethodPropertyNotFoundError** – If the estimator does not support Time Series Regression as a problem type.

**static load**(*file_path*)

Loads component at file path.

**Parameters** **file_path** (*str*) – Location to load file.

**Returns** ComponentBase object

**needs_fitting**(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need to be fit or whose fit methods do nothing.

**Returns** True.

**property parameters**(*self*)

Returns the parameters which were used to initialize the component.

**predict**(*self*, *X*)

Make predictions using the baseline regression strategy.

**Parameters** **X** (*pd.DataFrame*) – Data of shape [n_samples, n_features].

**Returns** Predicted values.

**Return type** pd.Series

**predict_proba**(*self*, *X: pandas.DataFrame*) → pandas.Series

Make probability estimates for labels.

**Parameters** **X** (*pd.DataFrame*) – Features.

**Returns** Probability estimates.

**Return type** pd.Series

**Raises** **MethodPropertyNotFoundError** – If estimator does not have a predict_proba method or a component_obj that implements predict_proba.

**save**(*self*, *file_path*, *pickle_protocol=cloudpickle.DEFAULT_PROTOCOL*)

Saves component at file path.

**Parameters**

- **file_path** (*str*) – Location to save file.
- **pickle_protocol** (*int*) – The pickle data stream format.

**update_parameters**(*self*, *update_dict*, *reset_fit=True*)

Updates the parameter dictionary of the component.

**Parameters**

- **update_dict** (*dict*) – A dict of parameters to update.
- **reset_fit** (*bool, optional*) – If True, will set *_is_fitted* to False.

### catboost_regressor

CatBoost Regressor, a regressor that uses gradient-boosting on decision trees. CatBoost is an open-source library and natively supports categorical features.

## Module Contents

## Classes Summary

| | |
|---|---|
| *CatBoostRegressor* | CatBoost Regressor, a regressor that uses gradient-boosting on decision trees. CatBoost is an open-source library and natively supports categorical features. |

## Contents

**class** evalml.pipelines.components.estimators.regressors.catboost_regressor.**CatBoostRegressor**(*n_estimators=*
*eta=0.03*,
*max_depth=6*
*boot-*
*strap_type=N*
*silent=False*,
*al-*
*low_writing_*
*ran-*
*dom_seed=0*,
*n_jobs=-*

*1*,
***kwargs*)

CatBoost Regressor, a regressor that uses gradient-boosting on decision trees. CatBoost is an open-source library and natively supports categorical features.

For more information, check out https://catboost.ai/

**Parameters**

- **n_estimators** (*float*) – The maximum number of trees to build. Defaults to 10.
- **eta** (*float*) – The learning rate. Defaults to 0.03.
- **max_depth** (*int*) – The maximum tree depth for base learners. Defaults to 6.
- **bootstrap_type** (*string*) – Defines the method for sampling the weights of objects. Available methods are 'Bayesian', 'Bernoulli', 'MVS'. Defaults to None.
- **silent** (*boolean*) – Whether to use the "silent" logging mode. Defaults to True.
- **allow_writing_files** (*boolean*) – Whether to allow writing snapshot files while training. Defaults to False.
- **n_jobs** (*int or None*) – Number of jobs to run in parallel. -1 uses all processes. Defaults to -1.
- **random_seed** (*int*) – Seed for the random number generator. Defaults to 0.

**Attributes**

| | |
|---|---|
| **hyper-parameter_ranges** | { "n_estimators": Integer(4, 100), "eta": Real(0.000001, 1), "max_depth": Integer(4, 10),} |
| **model_family** | ModelFamily.CATBOOST |
| **modifies_features** | True |
| **modifies_target** | False |
| **name** | CatBoost Regressor |
| **supported_problem_types** | [ ProblemTypes.REGRESSION, ProblemTypes.TIME_SERIES_REGRESSION,] |
| **training_only** | False |

**Methods**

| | |
|---|---|
| *clone* | Constructs a new component with the same parameters and random state. |
| *default_parameters* | Returns the default parameters for this component. |
| *describe* | Describe a component and its parameters. |
| *feature_importance* | Feature importance of fitted CatBoost regressor. |
| *fit* | Fits CatBoost regressor component to data. |
| *get_prediction_intervals* | Find the prediction intervals using the fitted regressor. |
| *load* | Loads component at file path. |
| *needs_fitting* | Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances. |
| *parameters* | Returns the parameters which were used to initialize the component. |
| *predict* | Make predictions using the fitted CatBoost regressor. |
| *predict_proba* | Make probability estimates for labels. |
| *save* | Saves component at file path. |
| *update_parameters* | Updates the parameter dictionary of the component. |

**clone**(*self*)

Constructs a new component with the same parameters and random state.

**Returns** A new instance of this component with identical parameters and random state.

**default_parameters**(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

**Returns** Default parameters for this component.

**Return type** dict

**describe**(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

**Parameters**

- **print_name** (*bool, optional*) – whether to print name of component

- **return_dict** (*bool, optional*) – whether to return description as dictionary in the format {"name": name, "parameters": parameters}

**Returns** Returns dictionary if return_dict is True, else None.

**Return type** None or dict

**property feature_importance**(*self*)

Feature importance of fitted CatBoost regressor.

**fit**(*self*, *X*, *y=None*)

Fits CatBoost regressor component to data.

**Parameters**

- **X** (*pd.DataFrame*) – The input training data of shape [n_samples, n_features].
- **y** (*pd.Series*) – The target training data of length [n_samples].

**Returns** self

**get_prediction_intervals**(*self*, *X: pandas.DataFrame*, *y: Optional[pandas.Series] = None*, *coverage: List[float] = None*, *predictions: pandas.Series = None*) → Dict[str, pandas.Series]

Find the prediction intervals using the fitted regressor.

This function takes the predictions of the fitted estimator and calculates the rolling standard deviation across all predictions using a window size of 5. The lower and upper predictions are determined by taking the percent point (quantile) function of the lower tail probability at each bound multiplied by the rolling standard deviation.

**Parameters**

- **X** (*pd.DataFrame*) – Data of shape [n_samples, n_features].
- **y** (*pd.Series*) – Target data. Ignored.
- **coverage** (*list[float]*) – A list of floats between the values 0 and 1 that the upper and lower bounds of the prediction interval should be calculated for.
- **predictions** (*pd.Series*) – Optional list of predictions to use. If None, will generate predictions using *X*.

**Returns** Prediction intervals, keys are in the format {coverage}_lower or {coverage}_upper.

**Return type** dict

**Raises** **MethodPropertyNotFoundError** – If the estimator does not support Time Series Regression as a problem type.

**static load**(*file_path*)

Loads component at file path.

**Parameters** **file_path** (*str*) – Location to load file.

**Returns** ComponentBase object

**needs_fitting**(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need to be fit or whose fit methods do nothing.

**Returns** True.

**property parameters**(*self*)

Returns the parameters which were used to initialize the component.

**predict**(*self*, *X*)

Make predictions using the fitted CatBoost regressor.

**Parameters** **X** (*pd.DataFrame*) – Data of shape [n_samples, n_features].

**Returns** Predicted values.

**Return type** pd.DataFrame

**predict_proba**(*self*, *X: pandas.DataFrame*) → pandas.Series

Make probability estimates for labels.

**Parameters** **X** (*pd.DataFrame*) – Features.

**Returns** Probability estimates.

**Return type** pd.Series

**Raises** **MethodPropertyNotFoundError** – If estimator does not have a predict_proba method or a component_obj that implements predict_proba.

**save**(*self*, *file_path*, *pickle_protocol=cloudpickle.DEFAULT_PROTOCOL*)

Saves component at file path.

**Parameters**

- **file_path** (*str*) – Location to save file.
- **pickle_protocol** (*int*) – The pickle data stream format.

**update_parameters**(*self*, *update_dict*, *reset_fit=True*)

Updates the parameter dictionary of the component.

**Parameters**

- **update_dict** (*dict*) – A dict of parameters to update.
- **reset_fit** (*bool, optional*) – If True, will set *_is_fitted* to False.

## decision_tree_regressor

Decision Tree Regressor.

### Module Contents

### Classes Summary

| | |
|---|---|
| *DecisionTreeRegressor* | Decision Tree Regressor. |

## Contents

**class** evalml.pipelines.components.estimators.regressors.decision_tree_regressor.**DecisionTreeRegressor**(*c*
*m*
*m*
*m*
*m*
*r*
*d*
***

Decision Tree Regressor.

**Parameters**

- **criterion** (*{"squared_error", "friedman_mse", "absolute_error", "poisson"}*) – The function to measure the quality of a split. Supported criteria are:
  - "squared_error" for the mean squared error, which is equal to variance reduction as feature selection criterion and minimizes the L2 loss using the mean of each terminal node
  - "friedman_mse", which uses mean squared error with Friedman"s improvement score for potential splits
  - "absolute_error" for the mean absolute error, which minimizes the L1 loss using the median of each terminal node,
  - "poisson" which uses reduction in Poisson deviance to find splits.
- **max_features** (*int, float or {"auto", "sqrt", "log2"}*) – The number of features to consider when looking for the best split:
  - If int, then consider max_features features at each split.
  - If float, then max_features is a fraction and int(max_features * n_features) features are considered at each split.
  - If "auto", then max_features=sqrt(n_features).
  - If "sqrt", then max_features=sqrt(n_features).
  - If "log2", then max_features=log2(n_features).
  - If None, then max_features = n_features.

  The search for a split does not stop until at least one valid partition of the node samples is found, even if it requires to effectively inspect more than max_features features.
- **max_depth** (*int*) – The maximum depth of the tree. Defaults to 6.
- **min_samples_split** (*int or float*) – The minimum number of samples required to split an internal node:
  - If int, then consider min_samples_split as the minimum number.
  - If float, then min_samples_split is a fraction and ceil(min_samples_split * n_samples) are the minimum number of samples for each split.

  Defaults to 2.
- **min_weight_fraction_leaf** (*float*) – The minimum weighted fraction of the sum total of weights (of all the input samples) required to be at a leaf node. Defaults to 0.0.
- **random_seed** (*int*) – Seed for the random number generator. Defaults to 0.

**Attributes**

| | |
|---|---|
| **hyper-parameter_ranges** | { "criterion": ["squared_error", "friedman_mse", "absolute_error"], "max_features": ["auto", "sqrt", "log2"], "max_depth": Integer(4, 10),} |
| **model_family** | ModelFamily.DECISION_TREE |
| **modifies_features** | True |
| **modifies_target** | False |
| **name** | Decision Tree Regressor |
| **supported_problem_types** | [ ProblemTypes.REGRESSION, ProblemTypes.TIME_SERIES_REGRESSION,] |
| **training_only** | False |

**Methods**

| | |
|---|---|
| *clone* | Constructs a new component with the same parameters and random state. |
| *default_parameters* | Returns the default parameters for this component. |
| *describe* | Describe a component and its parameters. |
| *feature_importance* | Returns importance associated with each feature. |
| *fit* | Fits estimator to data. |
| *get_prediction_intervals* | Find the prediction intervals using the fitted regressor. |
| *load* | Loads component at file path. |
| *needs_fitting* | Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances. |
| *parameters* | Returns the parameters which were used to initialize the component. |
| *predict* | Make predictions using selected features. |
| *predict_proba* | Make probability estimates for labels. |
| *save* | Saves component at file path. |
| *update_parameters* | Updates the parameter dictionary of the component. |

**clone**(*self*)

Constructs a new component with the same parameters and random state.

**Returns** A new instance of this component with identical parameters and random state.

**default_parameters**(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

**Returns** Default parameters for this component.

**Return type** dict

**describe**(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

**Parameters**

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description as dictionary in the format {"name": name, "parameters": parameters}

**Returns** Returns dictionary if return_dict is True, else None.

**Return type** None or dict

**property feature_importance**(*self*) → pandas.Series

Returns importance associated with each feature.

**Returns** Importance associated with each feature.

**Return type** np.ndarray

**Raises** **MethodPropertyNotFoundError** – If estimator does not have a feature_importance method or a component_obj that implements feature_importance.

**fit**(*self, X: pandas.DataFrame, y: Optional[pandas.Series] = None*)

Fits estimator to data.

**Parameters**

- **X** (*pd.DataFrame*) – The input training data of shape [n_samples, n_features].
- **y** (*pd.Series, optional*) – The target training data of length [n_samples].

**Returns** self

**get_prediction_intervals**(*self, X: pandas.DataFrame, y: Optional[pandas.Series] = None, coverage: List[float] = None, predictions: pandas.Series = None*) → Dict[str, pandas.Series]

Find the prediction intervals using the fitted regressor.

This function takes the predictions of the fitted estimator and calculates the rolling standard deviation across all predictions using a window size of 5. The lower and upper predictions are determined by taking the percent point (quantile) function of the lower tail probability at each bound multiplied by the rolling standard deviation.

**Parameters**

- **X** (*pd.DataFrame*) – Data of shape [n_samples, n_features].
- **y** (*pd.Series*) – Target data. Ignored.
- **coverage** (*list[float]*) – A list of floats between the values 0 and 1 that the upper and lower bounds of the prediction interval should be calculated for.
- **predictions** (*pd.Series*) – Optional list of predictions to use. If None, will generate predictions using *X*.

**Returns** Prediction intervals, keys are in the format {coverage}_lower or {coverage}_upper.

**Return type** dict

**Raises** **MethodPropertyNotFoundError** – If the estimator does not support Time Series Regression as a problem type.

**static load**(*file_path*)

Loads component at file path.

**Parameters** **file_path** (*str*) – Location to load file.

**Returns** ComponentBase object

**needs_fitting**(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need to be fit or whose fit methods do nothing.

**Returns** True.

**property parameters**(*self*)

Returns the parameters which were used to initialize the component.

**predict**(*self*, *X: pandas.DataFrame*) → pandas.Series

Make predictions using selected features.

**Parameters** **X** (*pd.DataFrame*) – Data of shape [n_samples, n_features].

**Returns** Predicted values.

**Return type** pd.Series

**Raises** **MethodPropertyNotFoundError** – If estimator does not have a predict method or a component_obj that implements predict.

**predict_proba**(*self*, *X: pandas.DataFrame*) → pandas.Series

Make probability estimates for labels.

**Parameters** **X** (*pd.DataFrame*) – Features.

**Returns** Probability estimates.

**Return type** pd.Series

**Raises** **MethodPropertyNotFoundError** – If estimator does not have a predict_proba method or a component_obj that implements predict_proba.

**save**(*self*, *file_path*, *pickle_protocol=cloudpickle.DEFAULT_PROTOCOL*)

Saves component at file path.

**Parameters**

- **file_path** (*str*) – Location to save file.
- **pickle_protocol** (*int*) – The pickle data stream format.

**update_parameters**(*self*, *update_dict*, *reset_fit=True*)

Updates the parameter dictionary of the component.

**Parameters**

- **update_dict** (*dict*) – A dict of parameters to update.
- **reset_fit** (*bool, optional*) – If True, will set *_is_fitted* to False.

### elasticnet_regressor

Elastic Net Regressor.

### Module Contents

### Classes Summary

| | |
|---|---|
| *ElasticNetRegressor* | Elastic Net Regressor. |

### Contents

**class** evalml.pipelines.components.estimators.regressors.elasticnet_regressor.**ElasticNetRegressor**(*alpha=0 l1_ratio max_iter ran- dom_see **kwarg*

Elastic Net Regressor.

**Parameters**

- **alpha** (*float*) – Constant that multiplies the penalty terms. Defaults to 0.0001.
- **l1_ratio** (*float*) – The mixing parameter, with 0 <= l1_ratio <= 1. Only used if penalty='elasticnet'. Setting l1_ratio=0 is equivalent to using penalty='l2', while setting l1_ratio=1 is equivalent to using penalty='l1'. For 0 < l1_ratio <1, the penalty is a combination of L1 and L2. Defaults to 0.15.
- **max_iter** (*int*) – The maximum number of iterations. Defaults to 1000.
- **random_seed** (*int*) – Seed for the random number generator. Defaults to 0.

**Attributes**

| | |
|---|---|
| **hyper-parameter_ranges** | { "alpha": Real(0, 1), "l1_ratio": Real(0, 1),} |
| **model_family** | ModelFamily.LINEAR_MODEL |
| **modifies_features** | True |
| **modifies_target** | False |
| **name** | Elastic Net Regressor |
| **supported_problem_types** | [ ProblemTypes.REGRESSION, ProblemTypes.TIME_SERIES_REGRESSION,] |
| **training_only** | False |

**Methods**

| | |
|---|---|
| *clone* | Constructs a new component with the same parameters and random state. |
| *default_parameters* | Returns the default parameters for this component. |
| *describe* | Describe a component and its parameters. |
| *feature_importance* | Feature importance for fitted ElasticNet regressor. |
| *fit* | Fits estimator to data. |
| *get_prediction_intervals* | Find the prediction intervals using the fitted regressor. |
| *load* | Loads component at file path. |
| *needs_fitting* | Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances. |
| *parameters* | Returns the parameters which were used to initialize the component. |
| *predict* | Make predictions using selected features. |
| *predict_proba* | Make probability estimates for labels. |
| *save* | Saves component at file path. |
| *update_parameters* | Updates the parameter dictionary of the component. |

**clone**(*self*)

Constructs a new component with the same parameters and random state.

**Returns** A new instance of this component with identical parameters and random state.

**default_parameters**(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

**Returns** Default parameters for this component.

**Return type** dict

**describe**(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

**Parameters**

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description as dictionary in the format {"name": name, "parameters": parameters}

**Returns** Returns dictionary if return_dict is True, else None.

**Return type** None or dict

**property feature_importance**(*self*)

Feature importance for fitted ElasticNet regressor.

**fit**(*self*, *X: pandas.DataFrame*, *y: Optional[pandas.Series] = None*)

Fits estimator to data.

**Parameters**

- **X** (*pd.DataFrame*) – The input training data of shape [n_samples, n_features].
- **y** (*pd.Series, optional*) – The target training data of length [n_samples].

**Returns** self

**get_prediction_intervals**(*self*, *X: pandas.DataFrame*, *y: Optional[pandas.Series] = None*, *coverage: List[float] = None*, *predictions: pandas.Series = None*) → Dict[str, pandas.Series]

Find the prediction intervals using the fitted regressor.

This function takes the predictions of the fitted estimator and calculates the rolling standard deviation across all predictions using a window size of 5. The lower and upper predictions are determined by taking the percent point (quantile) function of the lower tail probability at each bound multiplied by the rolling standard deviation.

**Parameters**

- **X** (*pd.DataFrame*) – Data of shape [n_samples, n_features].
- **y** (*pd.Series*) – Target data. Ignored.
- **coverage** (*list[float]*) – A list of floats between the values 0 and 1 that the upper and lower bounds of the prediction interval should be calculated for.
- **predictions** (*pd.Series*) – Optional list of predictions to use. If None, will generate predictions using *X*.

**Returns** Prediction intervals, keys are in the format {coverage}_lower or {coverage}_upper.

**Return type** dict

**Raises** **MethodPropertyNotFoundError** – If the estimator does not support Time Series Regression as a problem type.

**static load**(*file_path*)

Loads component at file path.

**Parameters** **file_path** (*str*) – Location to load file.

**Returns** ComponentBase object

**needs_fitting**(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need to be fit or whose fit methods do nothing.

**Returns** True.

**property parameters**(*self*)

Returns the parameters which were used to initialize the component.

**predict**(*self*, *X: pandas.DataFrame*) → pandas.Series

Make predictions using selected features.

**Parameters** **X** (*pd.DataFrame*) – Data of shape [n_samples, n_features].

**Returns** Predicted values.

**Return type** pd.Series

**Raises** **MethodPropertyNotFoundError** – If estimator does not have a predict method or a component_obj that implements predict.

**predict_proba**(*self*, *X: pandas.DataFrame*) → pandas.Series

Make probability estimates for labels.

**Parameters** **X** (*pd.DataFrame*) – Features.

**Returns** Probability estimates.

**Return type** pd.Series

**Raises** `MethodPropertyNotFoundError` – If estimator does not have a predict_proba method or a component_obj that implements predict_proba.

`save`(*self*, *file_path*, *pickle_protocol=cloudpickle.DEFAULT_PROTOCOL*)

Saves component at file path.

**Parameters**

- `file_path` (*str*) – Location to save file.
- `pickle_protocol` (*int*) – The pickle data stream format.

`update_parameters`(*self*, *update_dict*, *reset_fit=True*)

Updates the parameter dictionary of the component.

**Parameters**

- `update_dict` (*dict*) – A dict of parameters to update.
- `reset_fit` (*bool, optional*) – If True, will set *_is_fitted* to False.

## et_regressor

Extra Trees Regressor.

### Module Contents

### Classes Summary

| | |
|---|---|
| *ExtraTreesRegressor* | Extra Trees Regressor. |

### Contents

**class** evalml.pipelines.components.estimators.regressors.et_regressor.**ExtraTreesRegressor**(*n_estimators: int = 100, max_features: str = 'auto', max_depth: int = 6, min_samples_split int = 2, min_weight_fractio float = 0.0, n_jobs: int = - 1, random_seed: Union[int, float] = 0, **kwargs*)

Extra Trees Regressor.

**Parameters**

- **n_estimators** (*float*) – The number of trees in the forest. Defaults to 100.
- **max_features** (*int, float or {"auto", "sqrt", "log2"}*) – The number of features to consider when looking for the best split:
  - If int, then consider max_features features at each split.
  - If float, then max_features is a fraction and int(max_features * n_features) features are considered at each split.
  - If "auto", then max_features=sqrt(n_features).
  - If "sqrt", then max_features=sqrt(n_features).
  - If "log2", then max_features=log2(n_features).
  - If None, then max_features = n_features.

  The search for a split does not stop until at least one valid partition of the node samples is found, even if it requires to effectively inspect more than max_features features. Defaults to "auto".

- **max_depth** (*int*) – The maximum depth of the tree. Defaults to 6.
- **min_samples_split** (*int or float*) – The minimum number of samples required to split an internal node:
  - If int, then consider min_samples_split as the minimum number.
  - If float, then min_samples_split is a fraction and ceil(min_samples_split * n_samples) are the minimum number of samples for each split.
- **2.** (*Defaults to*) –
- **min_weight_fraction_leaf** (*float*) – The minimum weighted fraction of the sum total of weights (of all the input samples) required to be at a leaf node. Defaults to 0.0.
- **n_jobs** (*int or None*) – Number of jobs to run in parallel. -1 uses all processes. Defaults to -1.
- **random_seed** (*int*) – Seed for the random number generator. Defaults to 0.

**Attributes**

| | |
|---|---|
| **hyper-parameter_ranges** | { "n_estimators": Integer(10, 1000), "max_features": ["auto", "sqrt", "log2"], "max_depth": Integer(4, 10),} |
| **model_family** | ModelFamily.EXTRA_TREES |
| **modifies_features** | True |
| **modifies_target** | False |
| **name** | Extra Trees Regressor |
| **supported_problem_types** | [ ProblemTypes.REGRESSION, ProblemTypes.TIME_SERIES_REGRESSION,] |
| **training_only** | False |

**Methods**

| | |
|---|---|
| *clone* | Constructs a new component with the same parameters and random state. |
| *default_parameters* | Returns the default parameters for this component. |
| *describe* | Describe a component and its parameters. |
| *feature_importance* | Returns importance associated with each feature. |
| *fit* | Fits estimator to data. |
| *get_prediction_intervals* | Find the prediction intervals using the fitted ExtraTreesRegressor. |
| *load* | Loads component at file path. |
| *needs_fitting* | Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances. |
| *parameters* | Returns the parameters which were used to initialize the component. |
| *predict* | Make predictions using selected features. |
| *predict_proba* | Make probability estimates for labels. |
| *save* | Saves component at file path. |
| *update_parameters* | Updates the parameter dictionary of the component. |

**clone**(*self*)

Constructs a new component with the same parameters and random state.

**Returns** A new instance of this component with identical parameters and random state.

**default_parameters**(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

**Returns** Default parameters for this component.

**Return type** dict

**describe**(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

**Parameters**

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description as dictionary in the format {"name": name, "parameters": parameters}

**Returns** Returns dictionary if return_dict is True, else None.

**Return type** None or dict

**property feature_importance**(*self*) → pandas.Series

Returns importance associated with each feature.

**Returns** Importance associated with each feature.

**Return type** np.ndarray

**Raises** **MethodPropertyNotFoundError** – If estimator does not have a feature_importance method or a component_obj that implements feature_importance.

**fit**(*self*, *X: pandas.DataFrame*, *y: Optional[pandas.Series] = None*)

Fits estimator to data.

**Parameters**

- **X** (*pd.DataFrame*) – The input training data of shape [n_samples, n_features].
- **y** (*pd.Series, optional*) – The target training data of length [n_samples].

**Returns** self

**get_prediction_intervals**(*self*, *X: pandas.DataFrame*, *y: Optional[pandas.Series] = None*, *coverage: List[float] = None*, *predictions: pandas.Series = None*) → Dict[str, pandas.Series]

Find the prediction intervals using the fitted ExtraTreesRegressor.

**Parameters**

- **X** (*pd.DataFrame*) – Data of shape [n_samples, n_features].
- **y** (*pd.Series*) – Target data. Optional.
- **coverage** (*list[float]*) – A list of floats between the values 0 and 1 that the upper and lower bounds of the prediction interval should be calculated for.
- **predictions** (*pd.Series*) – Optional list of predictions to use. If None, will generate predictions using *X*.

**Returns** Prediction intervals, keys are in the format {coverage}_lower or {coverage}_upper.

**Return type** dict

**static load**(*file_path*)

Loads component at file path.

**Parameters** **file_path** (*str*) – Location to load file.

**Returns** ComponentBase object

**needs_fitting**(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need to be fit or whose fit methods do nothing.

**Returns** True.

**property parameters**(*self*)

Returns the parameters which were used to initialize the component.

**predict**(*self*, *X: pandas.DataFrame*) → pandas.Series

Make predictions using selected features.

**Parameters** **X** (*pd.DataFrame*) – Data of shape [n_samples, n_features].

**Returns** Predicted values.

**Return type** pd.Series

**Raises** **MethodPropertyNotFoundError** – If estimator does not have a predict method or a component_obj that implements predict.

**predict_proba**(*self*, *X: pandas.DataFrame*) → pandas.Series

Make probability estimates for labels.

**Parameters** **X** (*pd.DataFrame*) – Features.

**Returns** Probability estimates.

**Return type** pd.Series

**Raises** **MethodPropertyNotFoundError** – If estimator does not have a predict_proba method or a component_obj that implements predict_proba.

**save**(*self*, *file_path*, *pickle_protocol=cloudpickle.DEFAULT_PROTOCOL*)

Saves component at file path.

**Parameters**

- **file_path** (*str*) – Location to save file.
- **pickle_protocol** (*int*) – The pickle data stream format.

**update_parameters**(*self*, *update_dict*, *reset_fit=True*)

Updates the parameter dictionary of the component.

**Parameters**

- **update_dict** (*dict*) – A dict of parameters to update.
- **reset_fit** (*bool, optional*) – If True, will set *_is_fitted* to False.

### exponential_smoothing_regressor

Holt-Winters Exponential Smoothing Forecaster.

### Module Contents

### Classes Summary

| | |
|---|---|
| *ExponentialSmoothingRegressor* | Holt-Winters Exponential Smoothing Forecaster. |

### Contents

**class** evalml.pipelines.components.estimators.regressors.exponential_smoothing_regressor.**ExponentialSmoo**

Holt-Winters Exponential Smoothing Forecaster.

Currently ExponentialSmoothingRegressor isn't supported via conda install. It's recommended that it be installed via PyPI.

**Parameters**

- **trend** (*str*) – Type of trend component. Defaults to None.
- **damped_trend** (*bool*) – If the trend component should be damped. Defaults to False.
- **seasonal** (*str*) – Type of seasonal component. Takes one of {"additive", None}. Can also be multiplicative if
- **0** (*none of the target data is*) –
- **None.** (*but AutoMLSearch wiill not tune for this. Defaults to*) –
- **sp** (*int*) – The number of seasonal periods to consider. Defaults to 2.
- **n_jobs** (*int or None*) – Non-negative integer describing level of parallelism used for pipelines. Defaults to -1.
- **random_seed** (*int*) – Seed for the random number generator. Defaults to 0.

**Attributes**

| | |
|---|---|
| **hyper-parameter_ranges** | { "trend": [None, "additive"], "damped_trend": [True, False], "seasonal": [None, "additive"], "sp": Integer(2, 8),} |
| **model_family** | ModelFamily.EXPONENTIAL_SMOOTHING |
| **modifies_features** | True |
| **modifies_target** | False |
| **name** | Exponential Smoothing Regressor |
| **supported_problem_types** | [ProblemTypes.TIME_SERIES_REGRESSION] |
| **training_only** | False |

**Methods**

| | |
|---|---|
| *clone* | Constructs a new component with the same parameters and random state. |
| *default_parameters* | Returns the default parameters for this component. |
| *describe* | Describe a component and its parameters. |
| *feature_importance* | Returns array of 0's with a length of 1 as feature_importance is not defined for Exponential Smoothing regressor. |
| *fit* | Fits Exponential Smoothing Regressor to data. |
| *get_prediction_intervals* | Find the prediction intervals using the fitted ExponentialSmoothingRegressor. |
| *load* | Loads component at file path. |
| *needs_fitting* | Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances. |
| *parameters* | Returns the parameters which were used to initialize the component. |
| *predict* | Make predictions using fitted Exponential Smoothing regressor. |
| *predict_proba* | Make probability estimates for labels. |
| *save* | Saves component at file path. |
| *update_parameters* | Updates the parameter dictionary of the component. |

**clone**(*self*)

Constructs a new component with the same parameters and random state.

**Returns** A new instance of this component with identical parameters and random state.

**default_parameters**(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

**Returns** Default parameters for this component.

**Return type** dict

**describe**(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

**Parameters**

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description as dictionary in the format {"name": name, "parameters": parameters}

**Returns** Returns dictionary if return_dict is True, else None.

**Return type** None or dict

**property feature_importance**(*self*) → pandas.Series

Returns array of 0's with a length of 1 as feature_importance is not defined for Exponential Smoothing regressor.

**fit**(*self*, *X: pandas.DataFrame*, *y: Optional[pandas.Series] = None*)

Fits Exponential Smoothing Regressor to data.

**Parameters**

- **X** (*pd.DataFrame*) – The input training data of shape [n_samples, n_features]. Ignored.
- **y** (*pd.Series*) – The target training data of length [n_samples].

**Returns** self

**Raises** **ValueError** – If y was not passed in.

**get_prediction_intervals**(*self*, *X: pandas.DataFrame*, *y: Optional[pandas.Series] = None*, *coverage: List[float] = None*, *predictions: pandas.Series = None*) → Dict[str, pandas.Series]

Find the prediction intervals using the fitted ExponentialSmoothingRegressor.

Calculates the prediction intervals by using a simulation of the time series following a specified state space model.

**Parameters**

- **X** (*pd.DataFrame*) – Data of shape [n_samples, n_features].
- **y** (*pd.Series*) – Target data. Optional.
- **coverage** (*List[float]*) – A list of floats between the values 0 and 1 that the upper and lower bounds of the prediction interval should be calculated for.
- **predictions** (*pd.Series*) – Not used for Exponential Smoothing regressor.

**Returns** Prediction intervals, keys are in the format {coverage}_lower or {coverage}_upper.

**Return type** dict

**static load**(*file_path*)

Loads component at file path.

**Parameters** **file_path** (*str*) – Location to load file.

**Returns** ComponentBase object

**needs_fitting**(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need to be fit or whose fit methods do nothing.

**Returns** True.

**property parameters**(*self*)

Returns the parameters which were used to initialize the component.

**predict**(*self*, *X: pandas.DataFrame*, *y: Optional[pandas.Series] = None*) → pandas.Series

Make predictions using fitted Exponential Smoothing regressor.

**Parameters**

- **X** (*pd.DataFrame*) – Data of shape [n_samples, n_features]. Ignored except to set forecast horizon.
- **y** (*pd.Series*) – Target data.

**Returns** Predicted values.

**Return type** pd.Series

**predict_proba**(*self*, *X: pandas.DataFrame*) → pandas.Series

Make probability estimates for labels.

**Parameters** **X** (*pd.DataFrame*) – Features.

**Returns** Probability estimates.

**Return type** pd.Series

**Raises** **MethodPropertyNotFoundError** – If estimator does not have a predict_proba method or a component_obj that implements predict_proba.

**save**(*self*, *file_path*, *pickle_protocol=cloudpickle.DEFAULT_PROTOCOL*)

Saves component at file path.

**Parameters**

- **file_path** (*str*) – Location to save file.
- **pickle_protocol** (*int*) – The pickle data stream format.

**update_parameters**(*self*, *update_dict*, *reset_fit=True*)

Updates the parameter dictionary of the component.

**Parameters**

- **update_dict** (*dict*) – A dict of parameters to update.
- **reset_fit** (*bool, optional*) – If True, will set *_is_fitted* to False.

### lightgbm_regressor

LightGBM Regressor.

### Module Contents

### Classes Summary

| | |
|---|---|
| *LightGBMRegressor* | LightGBM Regressor. |

### Contents

**class** evalml.pipelines.components.estimators.regressors.lightgbm_regressor.**LightGBMRegressor**(*boosting_type learning_rate=0.1, n_estimators= max_depth=0 num_leaves= min_child_sa bagging_fraction bagging_freq=0, n_jobs=-1, random_seed=0, **kwargs*)

LightGBM Regressor.

**Parameters**

- **boosting_type** (*string*) – Type of boosting to use. Defaults to "gbdt". - 'gbdt' uses traditional Gradient Boosting Decision Tree - "dart", uses Dropouts meet Multiple Additive Regression Trees - "goss", uses Gradient-based One-Side Sampling - "rf", uses Random Forest
- **learning_rate** (*float*) – Boosting learning rate. Defaults to 0.1.
- **n_estimators** (*int*) – Number of boosted trees to fit. Defaults to 100.
- **max_depth** (*int*) – Maximum tree depth for base learners, <=0 means no limit. Defaults to 0.
- **num_leaves** (*int*) – Maximum tree leaves for base learners. Defaults to 31.
- **min_child_samples** (*int*) – Minimum number of data needed in a child (leaf). Defaults to 20.
- **bagging_fraction** (*float*) – LightGBM will randomly select a subset of features on each iteration (tree) without resampling if this is smaller than 1.0. For example, if set to 0.8, LightGBM will select 80% of features before training each tree. This can be used to speed up training and deal with overfitting. Defaults to 0.9.

- **bagging_freq** (*int*) – Frequency for bagging. 0 means bagging is disabled. k means perform bagging at every k iteration. Every k-th iteration, LightGBM will randomly select bagging_fraction * 100 % of the data to use for the next k iterations. Defaults to 0.
- **n_jobs** (*int or None*) – Number of threads to run in parallel. -1 uses all threads. Defaults to -1.
- **random_seed** (*int*) – Seed for the random number generator. Defaults to 0.

**Attributes**

| | |
|---|---|
| **hyper-parameter_ranges** | { "learning_rate": Real(0.000001, 1), "boosting_type": ["gbdt", "dart", "goss", "rf"], "n_estimators": Integer(10, 100), "max_depth": Integer(0, 10), "num_leaves": Integer(2, 100), "min_child_samples": Integer(1, 100), "bagging_fraction": Real(0.000001, 1), "bagging_freq": Integer(0, 1),} |
| **model_family** | ModelFamily.LIGHTGBM |
| **modifies_features** | True |
| **modifies_target** | False |
| **name** | LightGBM Regressor |
| **SEED_MAX** | SEED_BOUNDS.max_bound |
| **SEED_MIN** | 0 |
| **supported_problem_types** | [ProblemTypes.REGRESSION] |
| **training_only** | False |

**Methods**

| | |
|---|---|
| *clone* | Constructs a new component with the same parameters and random state. |
| *default_parameters* | Returns the default parameters for this component. |
| *describe* | Describe a component and its parameters. |
| *feature_importance* | Returns importance associated with each feature. |
| *fit* | Fits LightGBM regressor to data. |
| *get_prediction_intervals* | Find the prediction intervals using the fitted regressor. |
| *load* | Loads component at file path. |
| *needs_fitting* | Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances. |
| *parameters* | Returns the parameters which were used to initialize the component. |
| *predict* | Make predictions using fitted LightGBM regressor. |
| *predict_proba* | Make probability estimates for labels. |
| *save* | Saves component at file path. |
| *update_parameters* | Updates the parameter dictionary of the component. |

**clone**(*self*)

Constructs a new component with the same parameters and random state.

**Returns** A new instance of this component with identical parameters and random state.

**default_parameters**(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

**Returns** Default parameters for this component.

**Return type** dict

**describe**(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

**Parameters**

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description as dictionary in the format {"name": name, "parameters": parameters}

**Returns** Returns dictionary if return_dict is True, else None.

**Return type** None or dict

**property feature_importance**(*self*) → pandas.Series

Returns importance associated with each feature.

**Returns** Importance associated with each feature.

**Return type** np.ndarray

**Raises** **MethodPropertyNotFoundError** – If estimator does not have a feature_importance method or a component_obj that implements feature_importance.

**fit**(*self*, *X*, *y=None*)

Fits LightGBM regressor to data.

**Parameters**

- **X** (*pd.DataFrame*) – The input training data of shape [n_samples, n_features].
- **y** (*pd.Series*) – The target training data of length [n_samples].

**Returns** self

**get_prediction_intervals**(*self*, *X: pandas.DataFrame*, *y: Optional[pandas.Series] = None*, *coverage: List[float] = None*, *predictions: pandas.Series = None*) → Dict[str, pandas.Series]

Find the prediction intervals using the fitted regressor.

This function takes the predictions of the fitted estimator and calculates the rolling standard deviation across all predictions using a window size of 5. The lower and upper predictions are determined by taking the percent point (quantile) function of the lower tail probability at each bound multiplied by the rolling standard deviation.

**Parameters**

- **X** (*pd.DataFrame*) – Data of shape [n_samples, n_features].
- **y** (*pd.Series*) – Target data. Ignored.
- **coverage** (*list[float]*) – A list of floats between the values 0 and 1 that the upper and lower bounds of the prediction interval should be calculated for.
- **predictions** (*pd.Series*) – Optional list of predictions to use. If None, will generate predictions using *X*.

**Returns** Prediction intervals, keys are in the format {coverage}_lower or {coverage}_upper.

**Return type** dict

**Raises** `MethodPropertyNotFoundError` – If the estimator does not support Time Series Regression as a problem type.

`static load`(*file_path*)

Loads component at file path.

**Parameters** `file_path` (*str*) – Location to load file.

**Returns** ComponentBase object

`needs_fitting`(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need to be fit or whose fit methods do nothing.

**Returns** True.

`property parameters`(*self*)

Returns the parameters which were used to initialize the component.

`predict`(*self*, *X*)

Make predictions using fitted LightGBM regressor.

**Parameters** `X` (*pd.DataFrame*) – Data of shape [n_samples, n_features].

**Returns** Predicted values.

**Return type** pd.Series

`predict_proba`(*self*, *X: pandas.DataFrame*) → pandas.Series

Make probability estimates for labels.

**Parameters** `X` (*pd.DataFrame*) – Features.

**Returns** Probability estimates.

**Return type** pd.Series

**Raises** `MethodPropertyNotFoundError` – If estimator does not have a predict_proba method or a component_obj that implements predict_proba.

`save`(*self*, *file_path*, *pickle_protocol=cloudpickle.DEFAULT_PROTOCOL*)

Saves component at file path.

**Parameters**

- `file_path` (*str*) – Location to save file.
- `pickle_protocol` (*int*) – The pickle data stream format.

`update_parameters`(*self*, *update_dict*, *reset_fit=True*)

Updates the parameter dictionary of the component.

**Parameters**

- `update_dict` (*dict*) – A dict of parameters to update.
- `reset_fit` (*bool, optional*) – If True, will set *_is_fitted* to False.

### linear_regressor

Linear Regressor.

### Module Contents

### Classes Summary

| | |
|---|---|
| *LinearRegressor* | Linear Regressor. |

### Contents

**class** evalml.pipelines.components.estimators.regressors.linear_regressor.**LinearRegressor**(*fit_intercept=True*, *n_jobs=-1*, *random_seed=0*, ***kwargs*)

Linear Regressor.

**Parameters**

- **fit_intercept** (*boolean*) – Whether to calculate the intercept for this model. If set to False, no intercept will be used in calculations (i.e. data is expected to be centered). Defaults to True.
- **n_jobs** (*int or None*) – Number of jobs to run in parallel. -1 uses all threads. Defaults to -1.
- **random_seed** (*int*) – Seed for the random number generator. Defaults to 0.

**Attributes**

| | |
|---|---|
| **hyperparameter_ranges** | { "fit_intercept": [True, False],} |
| **model_family** | ModelFamily.LINEAR_MODEL |
| **modifies_features** | True |
| **modifies_target** | False |
| **name** | Linear Regressor |
| **supported_problem_types** | [ ProblemTypes.REGRESSION, ProblemTypes.TIME_SERIES_REGRESSION,] |
| **training_only** | False |

**Methods**

| | |
|---|---|
| *clone* | Constructs a new component with the same parameters and random state. |
| *default_parameters* | Returns the default parameters for this component. |
| *describe* | Describe a component and its parameters. |
| *feature_importance* | Feature importance for fitted linear regressor. |
| *fit* | Fits estimator to data. |
| *get_prediction_intervals* | Find the prediction intervals using the fitted regressor. |
| *load* | Loads component at file path. |
| *needs_fitting* | Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances. |
| *parameters* | Returns the parameters which were used to initialize the component. |
| *predict* | Make predictions using selected features. |
| *predict_proba* | Make probability estimates for labels. |
| *save* | Saves component at file path. |
| *update_parameters* | Updates the parameter dictionary of the component. |

**clone**(*self*)

Constructs a new component with the same parameters and random state.

**Returns** A new instance of this component with identical parameters and random state.

**default_parameters**(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

**Returns** Default parameters for this component.

**Return type** dict

**describe**(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

**Parameters**

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description as dictionary in the format {"name": name, "parameters": parameters}

**Returns** Returns dictionary if return_dict is True, else None.

**Return type** None or dict

**property feature_importance**(*self*)

Feature importance for fitted linear regressor.

**fit**(*self*, *X: pandas.DataFrame*, *y: Optional[pandas.Series] = None*)

Fits estimator to data.

**Parameters**

- **X** (*pd.DataFrame*) – The input training data of shape [n_samples, n_features].
- **y** (*pd.Series, optional*) – The target training data of length [n_samples].

**Returns** self

**get_prediction_intervals**(*self*, *X: pandas.DataFrame*, *y: Optional[pandas.Series] = None*, *coverage: List[float] = None*, *predictions: pandas.Series = None*) → Dict[str, pandas.Series]

Find the prediction intervals using the fitted regressor.

This function takes the predictions of the fitted estimator and calculates the rolling standard deviation across all predictions using a window size of 5. The lower and upper predictions are determined by taking the percent point (quantile) function of the lower tail probability at each bound multiplied by the rolling standard deviation.

**Parameters**

- **X** (*pd.DataFrame*) – Data of shape [n_samples, n_features].
- **y** (*pd.Series*) – Target data. Ignored.
- **coverage** (*list[float]*) – A list of floats between the values 0 and 1 that the upper and lower bounds of the prediction interval should be calculated for.
- **predictions** (*pd.Series*) – Optional list of predictions to use. If None, will generate predictions using *X*.

**Returns** Prediction intervals, keys are in the format {coverage}_lower or {coverage}_upper.

**Return type** dict

**Raises** **MethodPropertyNotFoundError** – If the estimator does not support Time Series Regression as a problem type.

**static load**(*file_path*)

Loads component at file path.

**Parameters** **file_path** (*str*) – Location to load file.

**Returns** ComponentBase object

**needs_fitting**(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need to be fit or whose fit methods do nothing.

**Returns** True.

**property parameters**(*self*)

Returns the parameters which were used to initialize the component.

**predict**(*self*, *X: pandas.DataFrame*) → pandas.Series

Make predictions using selected features.

**Parameters** **X** (*pd.DataFrame*) – Data of shape [n_samples, n_features].

**Returns** Predicted values.

**Return type** pd.Series

**Raises** **MethodPropertyNotFoundError** – If estimator does not have a predict method or a component_obj that implements predict.

**predict_proba**(*self*, *X: pandas.DataFrame*) → pandas.Series

Make probability estimates for labels.

**Parameters** **X** (*pd.DataFrame*) – Features.

**Returns** Probability estimates.

**Return type** pd.Series

**Raises** `MethodPropertyNotFoundError` – If estimator does not have a predict_proba method or a component_obj that implements predict_proba.

`save`(*self*, *file_path*, *pickle_protocol=cloudpickle.DEFAULT_PROTOCOL*)

Saves component at file path.

**Parameters**

- `file_path` (*str*) – Location to save file.
- `pickle_protocol` (*int*) – The pickle data stream format.

`update_parameters`(*self*, *update_dict*, *reset_fit=True*)

Updates the parameter dictionary of the component.

**Parameters**

- `update_dict` (*dict*) – A dict of parameters to update.
- `reset_fit` (*bool, optional*) – If True, will set *_is_fitted* to False.

### prophet_regressor

Prophet is a procedure for forecasting time series data based on an additive model where non-linear trends are fit with yearly, weekly, and daily seasonality, plus holiday effects. It works best with time series that have strong seasonal effects and several seasons of historical data. Prophet is robust to missing data and shifts in the trend, and typically handles outliers well.

### Module Contents

### Classes Summary

| | |
|---|---|
| *ProphetRegressor* | Prophet is a procedure for forecasting time series data based on an additive model where non-linear trends are fit with yearly, weekly, and daily seasonality, plus holiday effects. It works best with time series that have strong seasonal effects and several seasons of historical data. Prophet is robust to missing data and shifts in the trend, and typically handles outliers well. |

### Contents

**class** evalml.pipelines.components.estimators.regressors.prophet_regressor.**ProphetRegressor**(*time_index: Optional[Hashable = None, changepoint_prior_sca float = 0.05, seasonality_prior_scale: int = 10, holidays_prior_scal int = 10, seasonality_mode: str = 'additive', stan_backend: str = 'CMDSTANPY', interval_width: float = 0.95, random_seed: Union[int, float] = 0, **kwargs*)

Prophet is a procedure for forecasting time series data based on an additive model where non-linear trends are fit with yearly, weekly, and daily seasonality, plus holiday effects. It works best with time series that have strong seasonal effects and several seasons of historical data. Prophet is robust to missing data and shifts in the trend, and typically handles outliers well.

More information here: https://facebook.github.io/prophet/

**Parameters**

- **time_index** (*str*) – Specifies the name of the column in X that provides the datetime objects. Defaults to None.
- **changepoint_prior_scale** (*float*) – Determines the strength of the sparse prior for fitting on rate changes. Increasing this value increases the flexibility of the trend. Defaults to 0.05.
- **seasonality_prior_scale** (*int*) – Similar to changepoint_prior_scale. Adjusts the extent to which the seasonality model will fit the data. Defaults to 10.
- **holidays_prior_scale** (*int*) – Similar to changepoint_prior_scale. Adjusts the extent to which holidays will fit the data. Defaults to 10.
- **seasonality_mode** (*str*) – Determines how this component fits the seasonality. Options are "additive" and "multiplicative". Defaults to "additive".
- **stan_backend** (*str*) – Determines the backend that should be used to run Prophet. Options are "CMDSTANPY" and "PYSTAN". Defaults to "CMDSTANPY".
- **interval_width** (*float*) – Determines the confidence of the prediction interval range when calling *get_prediction_intervals*. Accepts values in the range (0,1). Defaults to 0.95.
- **random_seed** (*int*) – Seed for the random number generator. Defaults to 0.

**Attributes**

| | |
|---|---|
| **hyper-parameter_ranges** | { "changepoint_prior_scale": Real(0.001, 0.5), "seasonality_prior_scale": Real(0.01, 10), "holidays_prior_scale": Real(0.01, 10), "seasonality_mode": ["additive", "multiplicative"],} |
| **model_family** | ModelFamily.PROPHET |
| **modifies_features** | True |
| **modifies_target** | False |
| **name** | Prophet Regressor |
| **supported_problem_types** | [ProblemTypes.TIME_SERIES_REGRESSION] |
| **training_only** | False |

**Methods**

| | |
|---|---|
| *build_prophet_df* | Build the Prophet data to pass fit and predict on. |
| *clone* | Constructs a new component with the same parameters and random state. |
| *default_parameters* | Returns the default parameters for this component. |
| *describe* | Describe a component and its parameters. |
| *feature_importance* | Returns array of 0's with len(1) as feature_importance is not defined for Prophet regressor. |
| *fit* | Fits Prophet regressor component to data. |
| *get_params* | Get parameters for the Prophet regressor. |
| *get_prediction_intervals* | Find the prediction intervals using the fitted ProphetRegressor. |
| *load* | Loads component at file path. |
| *needs_fitting* | Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances. |
| *parameters* | Returns the parameters which were used to initialize the component. |
| *predict* | Make predictions using fitted Prophet regressor. |
| *predict_proba* | Make probability estimates for labels. |
| *save* | Saves component at file path. |
| *update_parameters* | Updates the parameter dictionary of the component. |

**static build_prophet_df**(*X: pandas.DataFrame*, *y: Optional[pandas.Series] = None*, *time_index: str = 'ds'*) → pandas.DataFrame

Build the Prophet data to pass fit and predict on.

**clone**(*self*)

Constructs a new component with the same parameters and random state.

**Returns** A new instance of this component with identical parameters and random state.

**default_parameters**(*cls*) → dict

Returns the default parameters for this component.

**Returns** Default parameters for this component.

**Return type** dict

**describe**(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

**Parameters**

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description as dictionary in the format {"name": name, "parameters": parameters}

**Returns** Returns dictionary if return_dict is True, else None.

**Return type** None or dict

**property feature_importance**(*self*) → numpy.ndarray

Returns array of 0's with len(1) as feature_importance is not defined for Prophet regressor.

**fit**(*self*, *X: pandas.DataFrame*, *y: Optional[pandas.Series] = None*)

Fits Prophet regressor component to data.

**Parameters**

- **X** (*pd.DataFrame*) – The input training data of shape [n_samples, n_features].
- **y** (*pd.Series*) – The target training data of length [n_samples].

**Returns** self

**get_params**(*self*) → dict

Get parameters for the Prophet regressor.

**get_prediction_intervals**(*self*, *X: pandas.DataFrame*, *y: Optional[pandas.Series] = None*, *coverage: List[float] = None*, *predictions: pandas.Series = None*) → Dict[str, pandas.Series]

Find the prediction intervals using the fitted ProphetRegressor.

**Parameters**

- **X** (*pd.DataFrame*) – Data of shape [n_samples, n_features].
- **y** (*pd.Series*) – Target data. Ignored.
- **coverage** (*List[float]*) – A list of floats between the values 0 and 1 that the upper and lower bounds of the prediction interval should be calculated for.
- **predictions** (*pd.Series*) – Not used for Prophet estimator.

**Returns** Prediction intervals, keys are in the format {coverage}_lower or {coverage}_upper.

**Return type** dict

**static load**(*file_path*)

Loads component at file path.

**Parameters** **file_path** (*str*) – Location to load file.

**Returns** ComponentBase object

**needs_fitting**(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need to be fit or whose fit methods do nothing.

**Returns** True.

**property parameters**(*self*)

Returns the parameters which were used to initialize the component.

**predict**(*self*, *X: pandas.DataFrame*, *y: Optional[pandas.Series] = None*) → pandas.Series

Make predictions using fitted Prophet regressor.

**Parameters**

- **X** (*pd.DataFrame*) – Data of shape [n_samples, n_features].
- **y** (*pd.Series*) – Target data. Ignored.

**Returns** Predicted values.

**Return type** pd.Series

**predict_proba**(*self*, *X: pandas.DataFrame*) → pandas.Series

Make probability estimates for labels.

**Parameters** **X** (*pd.DataFrame*) – Features.

**Returns** Probability estimates.

**Return type** pd.Series

**Raises** **MethodPropertyNotFoundError** – If estimator does not have a predict_proba method or a component_obj that implements predict_proba.

**save**(*self*, *file_path*, *pickle_protocol=cloudpickle.DEFAULT_PROTOCOL*)

Saves component at file path.

**Parameters**

- **file_path** (*str*) – Location to save file.
- **pickle_protocol** (*int*) – The pickle data stream format.

**update_parameters**(*self*, *update_dict*, *reset_fit=True*)

Updates the parameter dictionary of the component.

**Parameters**

- **update_dict** (*dict*) – A dict of parameters to update.
- **reset_fit** (*bool, optional*) – If True, will set *_is_fitted* to False.

## rf_regressor

Random Forest Regressor.

### Module Contents

### Classes Summary

| | |
|---|---|
| *RandomForestRegressor* | Random Forest Regressor. |

### Contents

**class** evalml.pipelines.components.estimators.regressors.rf_regressor.**RandomForestRegressor**(*n_estimators: int = 100, max_depth: int = 6, n_jobs: int = -1, random_seed: Union[int, float] = 0, **kwargs*)

Random Forest Regressor.

**Parameters**

- **n_estimators** (*float*) – The number of trees in the forest. Defaults to 100.
- **max_depth** (*int*) – Maximum tree depth for base learners. Defaults to 6.
- **n_jobs** (*int or None*) – Number of jobs to run in parallel. -1 uses all processes. Defaults to -1.
- **random_seed** (*int*) – Seed for the random number generator. Defaults to 0.

**Attributes**

| | |
|---|---|
| **hyperparameter_ranges** | { "n_estimators": Integer(10, 1000), "max_depth": Integer(1, 32),} |
| **model_family** | ModelFamily.RANDOM_FOREST |
| **modifies_features** | True |
| **modifies_target** | False |
| **name** | Random Forest Regressor |
| **supported_problem_types** | [ ProblemTypes.REGRESSION, ProblemTypes.TIME_SERIES_REGRESSION,] |
| **training_only** | False |

**Methods**

| | |
|---|---|
| *clone* | Constructs a new component with the same parameters and random state. |
| *default_parameters* | Returns the default parameters for this component. |
| *describe* | Describe a component and its parameters. |
| *feature_importance* | Returns importance associated with each feature. |
| *fit* | Fits estimator to data. |
| *get_prediction_intervals* | Find the prediction intervals using the fitted RandomForestRegressor. |
| *load* | Loads component at file path. |
| *needs_fitting* | Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances. |
| *parameters* | Returns the parameters which were used to initialize the component. |
| *predict* | Make predictions using selected features. |
| *predict_proba* | Make probability estimates for labels. |
| *save* | Saves component at file path. |
| *update_parameters* | Updates the parameter dictionary of the component. |

**clone**(*self*)

Constructs a new component with the same parameters and random state.

**Returns** A new instance of this component with identical parameters and random state.

**default_parameters**(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

**Returns** Default parameters for this component.

**Return type** dict

**describe**(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

**Parameters**

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description as dictionary in the format {"name": name, "parameters": parameters}

**Returns** Returns dictionary if return_dict is True, else None.

**Return type** None or dict

**property feature_importance**(*self*) → pandas.Series

Returns importance associated with each feature.

**Returns** Importance associated with each feature.

**Return type** np.ndarray

**Raises** **MethodPropertyNotFoundError** – If estimator does not have a feature_importance method or a component_obj that implements feature_importance.

**fit**(*self*, *X: pandas.DataFrame*, *y: Optional[pandas.Series] = None*)

Fits estimator to data.

**Parameters**

- **X** (*pd.DataFrame*) – The input training data of shape [n_samples, n_features].
- **y** (*pd.Series, optional*) – The target training data of length [n_samples].

**Returns** self

**get_prediction_intervals**(*self, X: pandas.DataFrame, y: Optional[pandas.Series] = None, coverage: List[float] = None, predictions: pandas.Series = None*) → Dict[str, pandas.Series]

Find the prediction intervals using the fitted RandomForestRegressor.

**Parameters**

- **X** (*pd.DataFrame*) – Data of shape [n_samples, n_features].
- **y** (*pd.Series*) – Target data. Optional.
- **coverage** (*list[float]*) – A list of floats between the values 0 and 1 that the upper and lower bounds of the prediction interval should be calculated for.
- **predictions** (*pd.Series*) – Optional list of predictions to use. If None, will generate predictions using *X*.

**Returns** Prediction intervals, keys are in the format {coverage}_lower or {coverage}_upper.

**Return type** dict

**static load**(*file_path*)

Loads component at file path.

**Parameters** **file_path** (*str*) – Location to load file.

**Returns** ComponentBase object

**needs_fitting**(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need to be fit or whose fit methods do nothing.

**Returns** True.

**property parameters**(*self*)

Returns the parameters which were used to initialize the component.

**predict**(*self, X: pandas.DataFrame*) → pandas.Series

Make predictions using selected features.

**Parameters** **X** (*pd.DataFrame*) – Data of shape [n_samples, n_features].

**Returns** Predicted values.

**Return type** pd.Series

**Raises** **MethodPropertyNotFoundError** – If estimator does not have a predict method or a component_obj that implements predict.

**predict_proba**(*self, X: pandas.DataFrame*) → pandas.Series

Make probability estimates for labels.

**Parameters** **X** (*pd.DataFrame*) – Features.

**Returns** Probability estimates.

**Return type** pd.Series

**Raises** **MethodPropertyNotFoundError** – If estimator does not have a predict_proba method or a component_obj that implements predict_proba.

**save**(*self*, *file_path*, *pickle_protocol=cloudpickle.DEFAULT_PROTOCOL*)

Saves component at file path.

**Parameters**

- **file_path** (*str*) – Location to save file.
- **pickle_protocol** (*int*) – The pickle data stream format.

**update_parameters**(*self*, *update_dict*, *reset_fit=True*)

Updates the parameter dictionary of the component.

**Parameters**

- **update_dict** (*dict*) – A dict of parameters to update.
- **reset_fit** (*bool, optional*) – If True, will set *_is_fitted* to False.

## svm_regressor

Support Vector Machine Regressor.

### Module Contents

### Classes Summary

| | |
|---|---|
| *SVMRegressor* | Support Vector Machine Regressor. |

### Contents

**class** evalml.pipelines.components.estimators.regressors.svm_regressor.**SVMRegressor**(*C=1.0*, *kernel='rbf'*, *gamma='auto'*, *random_seed=0*, ***kwargs*)

Support Vector Machine Regressor.

**Parameters**

- **C** (*float*) – The regularization parameter. The strength of the regularization is inversely proportional to C. Must be strictly positive. The penalty is a squared l2 penalty. Defaults to 1.0.
- **kernel** (*{"poly", "rbf", "sigmoid"}*) – Specifies the kernel type to be used in the algorithm. Defaults to "rbf".
- **gamma** (*{"scale", "auto"} or float*) – Kernel coefficient for "rbf", "poly" and "sigmoid". Defaults to "auto". - If gamma='scale' is passed then it uses 1 / (n_features * X.var()) as value of gamma - If "auto" (default), uses 1 / n_features

- **random_seed** (*int*) – Seed for the random number generator. Defaults to 0.

**Attributes**

| | |
|---|---|
| **hyper-parameter_ranges** | { "C": Real(0, 10), "kernel": ["poly", "rbf", "sigmoid"], "gamma": ["scale", "auto"],} |
| **model_family** | ModelFamily.SVM |
| **modifies_features** | True |
| **modifies_target** | False |
| **name** | SVM Regressor |
| **supported_problem_types** | [ ProblemTypes.REGRESSION, ProblemTypes.TIME_SERIES_REGRESSION,] |
| **training_only** | False |

**Methods**

| | |
|---|---|
| *clone* | Constructs a new component with the same parameters and random state. |
| *default_parameters* | Returns the default parameters for this component. |
| *describe* | Describe a component and its parameters. |
| *feature_importance* | Feature importance of fitted SVM regresor. |
| *fit* | Fits estimator to data. |
| *get_prediction_intervals* | Find the prediction intervals using the fitted regressor. |
| *load* | Loads component at file path. |
| *needs_fitting* | Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances. |
| *parameters* | Returns the parameters which were used to initialize the component. |
| *predict* | Make predictions using selected features. |
| *predict_proba* | Make probability estimates for labels. |
| *save* | Saves component at file path. |
| *update_parameters* | Updates the parameter dictionary of the component. |

**clone**(*self*)

Constructs a new component with the same parameters and random state.

**Returns** A new instance of this component with identical parameters and random state.

**default_parameters**(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

**Returns** Default parameters for this component.

**Return type** dict

**describe**(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

**Parameters**

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description as dictionary in the format {"name": name, "parameters": parameters}

**Returns** Returns dictionary if return_dict is True, else None.

**Return type** None or dict

**property feature_importance**(*self*)

Feature importance of fitted SVM regresor.

Only works with linear kernels. If the kernel isn't linear, we return a numpy array of zeros.

**Returns** The feature importance of the fitted SVM regressor, or an array of zeroes if the kernel is not linear.

**fit**(*self, X: pandas.DataFrame, y: Optional[pandas.Series] = None*)

Fits estimator to data.

**Parameters**

- **X** (*pd.DataFrame*) – The input training data of shape [n_samples, n_features].
- **y** (*pd.Series, optional*) – The target training data of length [n_samples].

**Returns** self

**get_prediction_intervals**(*self, X: pandas.DataFrame, y: Optional[pandas.Series] = None, coverage: List[float] = None, predictions: pandas.Series = None*) → Dict[str, pandas.Series]

Find the prediction intervals using the fitted regressor.

This function takes the predictions of the fitted estimator and calculates the rolling standard deviation across all predictions using a window size of 5. The lower and upper predictions are determined by taking the percent point (quantile) function of the lower tail probability at each bound multiplied by the rolling standard deviation.

**Parameters**

- **X** (*pd.DataFrame*) – Data of shape [n_samples, n_features].
- **y** (*pd.Series*) – Target data. Ignored.
- **coverage** (*list[float]*) – A list of floats between the values 0 and 1 that the upper and lower bounds of the prediction interval should be calculated for.
- **predictions** (*pd.Series*) – Optional list of predictions to use. If None, will generate predictions using *X*.

**Returns** Prediction intervals, keys are in the format {coverage}_lower or {coverage}_upper.

**Return type** dict

**Raises** **MethodPropertyNotFoundError** – If the estimator does not support Time Series Regression as a problem type.

**static load**(*file_path*)

Loads component at file path.

**Parameters** **file_path** (*str*) – Location to load file.

**Returns** ComponentBase object

**needs_fitting**(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need to be fit or whose fit methods do nothing.

**Returns** True.

**property parameters**(*self*)

Returns the parameters which were used to initialize the component.

**predict**(*self*, *X: pandas.DataFrame*) → pandas.Series

Make predictions using selected features.

**Parameters** **X** (*pd.DataFrame*) – Data of shape [n_samples, n_features].

**Returns** Predicted values.

**Return type** pd.Series

**Raises** **MethodPropertyNotFoundError** – If estimator does not have a predict method or a component_obj that implements predict.

**predict_proba**(*self*, *X: pandas.DataFrame*) → pandas.Series

Make probability estimates for labels.

**Parameters** **X** (*pd.DataFrame*) – Features.

**Returns** Probability estimates.

**Return type** pd.Series

**Raises** **MethodPropertyNotFoundError** – If estimator does not have a predict_proba method or a component_obj that implements predict_proba.

**save**(*self*, *file_path*, *pickle_protocol=cloudpickle.DEFAULT_PROTOCOL*)

Saves component at file path.

**Parameters**

- **file_path** (*str*) – Location to save file.
- **pickle_protocol** (*int*) – The pickle data stream format.

**update_parameters**(*self*, *update_dict*, *reset_fit=True*)

Updates the parameter dictionary of the component.

**Parameters**

- **update_dict** (*dict*) – A dict of parameters to update.
- **reset_fit** (*bool, optional*) – If True, will set *_is_fitted* to False.

### time_series_baseline_estimator

Time series estimator that predicts using the naive forecasting approach.

### Module Contents

### Classes Summary

| | |
|---|---|
| *TimeSeriesBaselineEstimator* | Time series estimator that predicts using the naive forecasting approach. |

### Contents

**class** evalml.pipelines.components.estimators.regressors.time_series_baseline_estimator.**TimeSeriesBaseli**

Time series estimator that predicts using the naive forecasting approach.

This is useful as a simple baseline estimator for time series problems.

**Parameters**

- **gap** (*int*) – Gap between prediction date and target date and must be a positive integer. If gap is 0, target date will be shifted ahead by 1 time period. Defaults to 1.
- **forecast_horizon** (*int*) – Number of time steps the model is expected to predict.
- **random_seed** (*int*) – Seed for the random number generator. Defaults to 0.

**Attributes**

| | |
|---|---|
| **hyperparameter_ranges** | {} |
| **model_family** | ModelFamily.BASELINE |
| **modifies_features** | True |
| **modifies_target** | False |
| **name** | Time Series Baseline Estimator |
| **supported_problem_types** | [ ProblemTypes.TIME_SERIES_REGRESSION, ProblemTypes.TIME_SERIES_BINARY, ProblemTypes.TIME_SERIES_MULTICLASS,] |
| **training_only** | False |

**Methods**

| | |
|---|---|
| *clone* | Constructs a new component with the same parameters and random state. |
| *default_parameters* | Returns the default parameters for this component. |
| *describe* | Describe a component and its parameters. |
| *feature_importance* | Returns importance associated with each feature. |
| *fit* | Fits time series baseline estimator to data. |
| *get_prediction_intervals* | Find the prediction intervals using the fitted regressor. |
| *load* | Loads component at file path. |
| *needs_fitting* | Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances. |
| *parameters* | Returns the parameters which were used to initialize the component. |
| *predict* | Make predictions using fitted time series baseline estimator. |
| *predict_proba* | Make prediction probabilities using fitted time series baseline estimator. |
| *save* | Saves component at file path. |
| *update_parameters* | Updates the parameter dictionary of the component. |

**clone**(*self*)

Constructs a new component with the same parameters and random state.

**Returns** A new instance of this component with identical parameters and random state.

**default_parameters**(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

**Returns** Default parameters for this component.

**Return type** dict

**describe**(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

**Parameters**

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description as dictionary in the format {"name": name, "parameters": parameters}

**Returns** Returns dictionary if return_dict is True, else None.

**Return type** None or dict

**property feature_importance**(*self*)

Returns importance associated with each feature.

Since baseline estimators do not use input features to calculate predictions, returns an array of zeroes.

**Returns** An array of zeroes.

**Return type** np.ndarray (float)

**fit**(*self*, *X*, *y=None*)

Fits time series baseline estimator to data.

**Parameters**

- **X** (*pd.DataFrame*) – The input training data of shape [n_samples, n_features].
- **y** (*pd.Series*) – The target training data of length [n_samples].

**Returns** self

**Raises** **ValueError** – If input y is None.

**get_prediction_intervals**(*self*, *X: pandas.DataFrame*, *y: Optional[pandas.Series] = None*, *coverage: List[float] = None*, *predictions: pandas.Series = None*) → Dict[str, pandas.Series]

Find the prediction intervals using the fitted regressor.

This function takes the predictions of the fitted estimator and calculates the rolling standard deviation across all predictions using a window size of 5. The lower and upper predictions are determined by taking the percent point (quantile) function of the lower tail probability at each bound multiplied by the rolling standard deviation.

**Parameters**

- **X** (*pd.DataFrame*) – Data of shape [n_samples, n_features].
- **y** (*pd.Series*) – Target data. Ignored.
- **coverage** (*list[float]*) – A list of floats between the values 0 and 1 that the upper and lower bounds of the prediction interval should be calculated for.
- **predictions** (*pd.Series*) – Optional list of predictions to use. If None, will generate predictions using *X*.

**Returns** Prediction intervals, keys are in the format {coverage}_lower or {coverage}_upper.

**Return type** dict

**Raises** **MethodPropertyNotFoundError** – If the estimator does not support Time Series Regression as a problem type.

**static load**(*file_path*)

Loads component at file path.

**Parameters** **file_path** (*str*) – Location to load file.

**Returns** ComponentBase object

**needs_fitting**(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need to be fit or whose fit methods do nothing.

**Returns** True.

**property parameters**(*self*)

Returns the parameters which were used to initialize the component.

**predict**(*self*, *X*)

Make predictions using fitted time series baseline estimator.

**Parameters** **X** (*pd.DataFrame*) – Data of shape [n_samples, n_features].

**Returns** Predicted values.

**Return type** pd.Series

**Raises** `ValueError` – If input y is None.

`predict_proba`(*self*, *X*)

Make prediction probabilities using fitted time series baseline estimator.

**Parameters** `X` (*pd.DataFrame*) – Data of shape [n_samples, n_features].

**Returns** Predicted probability values.

**Return type** pd.DataFrame

**Raises** `ValueError` – If input y is None.

`save`(*self*, *file_path*, *pickle_protocol=cloudpickle.DEFAULT_PROTOCOL*)

Saves component at file path.

**Parameters**

- `file_path` (*str*) – Location to save file.
- `pickle_protocol` (*int*) – The pickle data stream format.

`update_parameters`(*self*, *update_dict*, *reset_fit=True*)

Updates the parameter dictionary of the component.

**Parameters**

- `update_dict` (*dict*) – A dict of parameters to update.
- `reset_fit` (*bool, optional*) – If True, will set *_is_fitted* to False.

### vowpal_wabbit_regressor

Vowpal Wabbit Regressor.

### Module Contents

### Classes Summary

| | |
|---|---|
| *VowpalWabbitRegressor* | Vowpal Wabbit Regressor. |

### Contents

**class** evalml.pipelines.components.estimators.regressors.vowpal_wabbit_regressor.**VowpalWabbitRegressor**(*le*
*d*
*c*
*p*
*p*
*r*
*d*
***

Vowpal Wabbit Regressor.

**Parameters**

- **learning_rate** (*float*) – Boosting learning rate. Defaults to 0.5.
- **decay_learning_rate** (*float*) – Decay factor for learning_rate. Defaults to 1.0.
- **power_t** (*float*) – Power on learning rate decay. Defaults to 0.5.
- **passes** (*int*) – Number of training passes. Defaults to 1.
- **random_seed** (*int*) – Seed for the random number generator. Defaults to 0.

**Attributes**

| | |
|---|---|
| **hyper-parameter_ranges** | None |
| **model_family** | ModelFamily.VOWPAL_WABBIT |
| **modifies_features** | True |
| **modifies_target** | False |
| **name** | Vowpal Wabbit Regressor |
| **supported_problem_types** | [ ProblemTypes.REGRESSION, ProblemTypes.TIME_SERIES_REGRESSION,] |
| **training_only** | False |

**Methods**

| | |
|---|---|
| *clone* | Constructs a new component with the same parameters and random state. |
| *default_parameters* | Returns the default parameters for this component. |
| *describe* | Describe a component and its parameters. |
| *feature_importance* | Feature importance for Vowpal Wabbit regressor. |
| *fit* | Fits estimator to data. |
| *get_prediction_intervals* | Find the prediction intervals using the fitted regressor. |
| *load* | Loads component at file path. |
| *needs_fitting* | Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances. |
| *parameters* | Returns the parameters which were used to initialize the component. |
| *predict* | Make predictions using selected features. |
| *predict_proba* | Make probability estimates for labels. |
| *save* | Saves component at file path. |
| *update_parameters* | Updates the parameter dictionary of the component. |

**clone**(*self*)

Constructs a new component with the same parameters and random state.

**Returns** A new instance of this component with identical parameters and random state.

**default_parameters**(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

**Returns** Default parameters for this component.

**Return type** dict

**describe**(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

**Parameters**

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description as dictionary in the format {"name": name, "parameters": parameters}

**Returns** Returns dictionary if return_dict is True, else None.

**Return type** None or dict

**property feature_importance**(*self*)

Feature importance for Vowpal Wabbit regressor.

**fit**(*self*, *X: pandas.DataFrame*, *y: Optional[pandas.Series] = None*)

Fits estimator to data.

**Parameters**

- **X** (*pd.DataFrame*) – The input training data of shape [n_samples, n_features].
- **y** (*pd.Series, optional*) – The target training data of length [n_samples].

**Returns** self

**get_prediction_intervals**(*self*, *X: pandas.DataFrame*, *y: Optional[pandas.Series] = None*, *coverage: List[float] = None*, *predictions: pandas.Series = None*) → Dict[str, pandas.Series]

Find the prediction intervals using the fitted regressor.

This function takes the predictions of the fitted estimator and calculates the rolling standard deviation across all predictions using a window size of 5. The lower and upper predictions are determined by taking the percent point (quantile) function of the lower tail probability at each bound multiplied by the rolling standard deviation.

**Parameters**

- **X** (*pd.DataFrame*) – Data of shape [n_samples, n_features].
- **y** (*pd.Series*) – Target data. Ignored.
- **coverage** (*list[float]*) – A list of floats between the values 0 and 1 that the upper and lower bounds of the prediction interval should be calculated for.
- **predictions** (*pd.Series*) – Optional list of predictions to use. If None, will generate predictions using *X*.

**Returns** Prediction intervals, keys are in the format {coverage}_lower or {coverage}_upper.

**Return type** dict

**Raises** **MethodPropertyNotFoundError** – If the estimator does not support Time Series Regression as a problem type.

**static load**(*file_path*)

Loads component at file path.

**Parameters** **file_path** (*str*) – Location to load file.

**Returns** ComponentBase object

**needs_fitting**(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need to be fit or whose fit methods do nothing.

**Returns** True.

**property parameters**(*self*)

Returns the parameters which were used to initialize the component.

**predict**(*self*, *X: pandas.DataFrame*) → pandas.Series

Make predictions using selected features.

**Parameters** **X** (*pd.DataFrame*) – Data of shape [n_samples, n_features].

**Returns** Predicted values.

**Return type** pd.Series

**Raises** **MethodPropertyNotFoundError** – If estimator does not have a predict method or a component_obj that implements predict.

**predict_proba**(*self*, *X: pandas.DataFrame*) → pandas.Series

Make probability estimates for labels.

**Parameters** **X** (*pd.DataFrame*) – Features.

**Returns** Probability estimates.

**Return type** pd.Series

**Raises** **MethodPropertyNotFoundError** – If estimator does not have a predict_proba method or a component_obj that implements predict_proba.

**save**(*self*, *file_path*, *pickle_protocol=cloudpickle.DEFAULT_PROTOCOL*)

Saves component at file path.

**Parameters**

- **file_path** (*str*) – Location to save file.
- **pickle_protocol** (*int*) – The pickle data stream format.

**update_parameters**(*self*, *update_dict*, *reset_fit=True*)

Updates the parameter dictionary of the component.

**Parameters**

- **update_dict** (*dict*) – A dict of parameters to update.
- **reset_fit** (*bool, optional*) – If True, will set *_is_fitted* to False.

### xgboost_regressor

XGBoost Regressor.

### Module Contents

### Classes Summary

| | |
|---|---|
| *XGBoostRegressor* | XGBoost Regressor. |

### Contents

**class** evalml.pipelines.components.estimators.regressors.xgboost_regressor.**XGBoostRegressor**(*eta: float = 0.1, max_depth: int = 6, min_child_weig int = 1, n_estimators: int = 100, random_seed: Union[int, float] = 0, n_jobs: int = 12, **kwargs*)

XGBoost Regressor.

**Parameters**

- **eta** (*float*) – Boosting learning rate. Defaults to 0.1.
- **max_depth** (*int*) – Maximum tree depth for base learners. Defaults to 6.
- **min_child_weight** (*float*) – Minimum sum of instance weight (hessian) needed in a child. Defaults to 1.0
- **n_estimators** (*int*) – Number of gradient boosted trees. Equivalent to number of boosting rounds. Defaults to 100.

- **random_seed** (*int*) – Seed for the random number generator. Defaults to 0.
- **n_jobs** (*int*) – Number of parallel threads used to run xgboost. Note that creating thread contention will significantly slow down the algorithm. Defaults to 12.

**Attributes**

| | |
|---|---|
| **hyper-parameter_ranges** | { "eta": Real(0.000001, 1), "max_depth": Integer(1, 20), "min_child_weight": Real(1, 10), "n_estimators": Integer(1, 1000),} |
| **model_family** | ModelFamily.XGBOOST |
| **modifies_features** | True |
| **modifies_target** | False |
| **name** | XGBoost Regressor |
| **SEED_MAX** | None |
| **SEED_MIN** | None |
| **supported_problem_types** | [ ProblemTypes.REGRESSION, ProblemTypes.TIME_SERIES_REGRESSION,] |
| **training_only** | False |

**Methods**

| | |
|---|---|
| *clone* | Constructs a new component with the same parameters and random state. |
| *default_parameters* | Returns the default parameters for this component. |
| *describe* | Describe a component and its parameters. |
| *feature_importance* | Feature importance of fitted XGBoost regressor. |
| *fit* | Fits XGBoost regressor component to data. |
| *get_prediction_intervals* | Find the prediction intervals using the fitted XGBoostRegressor. |
| *load* | Loads component at file path. |
| *needs_fitting* | Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances. |
| *parameters* | Returns the parameters which were used to initialize the component. |
| *predict* | Make predictions using fitted XGBoost regressor. |
| *predict_proba* | Make probability estimates for labels. |
| *save* | Saves component at file path. |
| *update_parameters* | Updates the parameter dictionary of the component. |

**clone**(*self*)

Constructs a new component with the same parameters and random state.

**Returns** A new instance of this component with identical parameters and random state.

**default_parameters**(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

**Returns** Default parameters for this component.

**Return type** dict

**describe**(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

**Parameters**

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description as dictionary in the format {"name": name, "parameters": parameters}

**Returns** Returns dictionary if return_dict is True, else None.

**Return type** None or dict

**property feature_importance**(*self*) → pandas.Series

Feature importance of fitted XGBoost regressor.

**fit**(*self*, *X: pandas.DataFrame*, *y: Optional[pandas.Series] = None*)

Fits XGBoost regressor component to data.

**Parameters**

- **X** (*pd.DataFrame*) – The input training data of shape [n_samples, n_features].
- **y** (*pd.Series, optional*) – The target training data of length [n_samples].

**Returns** self

**get_prediction_intervals**(*self*, *X: pandas.DataFrame*, *y: Optional[pandas.Series] = None*, *coverage: List[float] = None*, *predictions: pandas.Series = None*) → Dict[str, pandas.Series]

Find the prediction intervals using the fitted XGBoostRegressor.

**Parameters**

- **X** (*pd.DataFrame*) – Data of shape [n_samples, n_features].
- **y** (*pd.Series*) – Target data. Ignored.
- **coverage** (*List[float]*) – A list of floats between the values 0 and 1 that the upper and lower bounds of the prediction interval should be calculated for.
- **predictions** (*pd.Series*) – Optional list of predictions to use. If None, will generate predictions using *X*.

**Returns** Prediction intervals, keys are in the format {coverage}_lower or {coverage}_upper.

**Return type** dict

**static load**(*file_path*)

Loads component at file path.

**Parameters** **file_path** (*str*) – Location to load file.

**Returns** ComponentBase object

**needs_fitting**(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need to be fit or whose fit methods do nothing.

**Returns** True.

**property parameters**(*self*)

Returns the parameters which were used to initialize the component.

**predict**(*self*, *X: pandas.DataFrame*) → pandas.Series

Make predictions using fitted XGBoost regressor.

**Parameters** **X** (*pd.DataFrame*) – Data of shape [n_samples, n_features].

**Returns** Predicted values.

**Return type** pd.Series

**predict_proba**(*self*, *X: pandas.DataFrame*) → pandas.Series

Make probability estimates for labels.

**Parameters** **X** (*pd.DataFrame*) – Features.

**Returns** Probability estimates.

**Return type** pd.Series

**Raises** **MethodPropertyNotFoundError** – If estimator does not have a predict_proba method or a component_obj that implements predict_proba.

**save**(*self*, *file_path*, *pickle_protocol=cloudpickle.DEFAULT_PROTOCOL*)

Saves component at file path.

**Parameters**

- **file_path** (*str*) – Location to save file.
- **pickle_protocol** (*int*) – The pickle data stream format.

**update_parameters**(*self*, *update_dict*, *reset_fit=True*)

Updates the parameter dictionary of the component.

**Parameters**

- **update_dict** (*dict*) – A dict of parameters to update.
- **reset_fit** (*bool, optional*) – If True, will set *_is_fitted* to False.

### Package Contents

### Classes Summary

| | |
|---|---|
| *ARIMARegressor* | Autoregressive Integrated Moving Average Model. The three parameters (p, d, q) are the AR order, the degree of differencing, and the MA order. More information here: https://www.statsmodels.org/devel/generated/statsmodels.tsa.arima.model.ARIMA.html. |
| *BaselineRegressor* | Baseline regressor that uses a simple strategy to make predictions. This is useful as a simple baseline regressor to compare with other regressors. |
| *CatBoostRegressor* | CatBoost Regressor, a regressor that uses gradient-boosting on decision trees. CatBoost is an open-source library and natively supports categorical features. |
| *DecisionTreeRegressor* | Decision Tree Regressor. |
| *ElasticNetRegressor* | Elastic Net Regressor. |
| *ExponentialSmoothingRegressor* | Holt-Winters Exponential Smoothing Forecaster. |
| *ExtraTreesRegressor* | Extra Trees Regressor. |
| *LightGBMRegressor* | LightGBM Regressor. |
| *LinearRegressor* | Linear Regressor. |
| *ProphetRegressor* | Prophet is a procedure for forecasting time series data based on an additive model where non-linear trends are fit with yearly, weekly, and daily seasonality, plus holiday effects. It works best with time series that have strong seasonal effects and several seasons of historical data. Prophet is robust to missing data and shifts in the trend, and typically handles outliers well. |
| *RandomForestRegressor* | Random Forest Regressor. |
| *SVMRegressor* | Support Vector Machine Regressor. |
| *TimeSeriesBaselineEstimator* | Time series estimator that predicts using the naive forecasting approach. |
| *VowpalWabbitRegressor* | Vowpal Wabbit Regressor. |
| *XGBoostRegressor* | XGBoost Regressor. |

### Contents

**class** evalml.pipelines.components.estimators.regressors.**ARIMARegressor**(*time_index: Optional[Hashable] = None, trend: Optional[str] = None, start_p: int = 2, d: int = 0, start_q: int = 2, max_p: int = 5, max_d: int = 2, max_q: int = 5, seasonal: bool = True, sp: int = 1, n_jobs: int = - 1, random_seed: Union[int, float] = 0, maxiter: int = 10, use_covariates: bool = True, **kwargs*)

Autoregressive Integrated Moving Average Model. The three parameters (p, d, q) are the AR order, the degree of differencing, and the MA order. More information here: https://www.statsmodels.org/devel/generated/statsmodels.tsa.arima.model.ARIMA.html.

Currently ARIMARegressor isn't supported via conda install. It's recommended that it be installed via PyPI.

**Parameters**

- **time_index** (*str*) – Specifies the name of the column in X that provides the datetime objects. Defaults to None.
- **trend** (*str*) – Controls the deterministic trend. Options are ['n', 'c', 't', 'ct'] where 'c' is a constant term, 't' indicates a linear trend, and 'ct' is both. Can also be an iterable when defining a polynomial, such as [1, 1, 0, 1].
- **start_p** (*int*) – Minimum Autoregressive order. Defaults to 2.
- **d** (*int*) – Minimum Differencing degree. Defaults to 0.
- **start_q** (*int*) – Minimum Moving Average order. Defaults to 2.
- **max_p** (*int*) – Maximum Autoregressive order. Defaults to 5.
- **max_d** (*int*) – Maximum Differencing degree. Defaults to 2.
- **max_q** (*int*) – Maximum Moving Average order. Defaults to 5.
- **seasonal** (*boolean*) – Whether to fit a seasonal model to ARIMA. Defaults to True.
- **sp** (*int or str*) – Period for seasonal differencing, specifically the number of periods in each season. If "detect", this model will automatically detect this parameter (given the time series is a standard frequency) and will fall back to 1 (no seasonality) if it cannot be detected. Defaults to 1.
- **n_jobs** (*int or None*) – Non-negative integer describing level of parallelism used for pipelines. Defaults to -1.
- **random_seed** (*int*) – Seed for the random number generator. Defaults to 0.

**Attributes**

| | |
|---|---|
| **hyper-parameter_ranges** | { "start_p": Integer(1, 3), "d": Integer(0, 2), "start_q": Integer(1, 3), "max_p": Integer(3, 10), "max_d": Integer(2, 5), "max_q": Integer(3, 10), "seasonal": [True, False],} |
| **max_cols** | 7 |
| **max_rows** | 1000 |
| **model_family** | ModelFamily.ARIMA |
| **modifies_features** | True |
| **modifies_target** | False |
| **name** | ARIMA Regressor |
| **supported_problem_types** | [ProblemTypes.TIME_SERIES_REGRESSION] |
| **training_only** | False |

**Methods**

| | |
|---|---|
| *clone* | Constructs a new component with the same parameters and random state. |
| *default_parameters* | Returns the default parameters for this component. |
| *describe* | Describe a component and its parameters. |
| *feature_importance* | Returns array of 0's with a length of 1 as feature_importance is not defined for ARIMA regressor. |
| *fit* | Fits ARIMA regressor to data. |
| *get_prediction_intervals* | Find the prediction intervals using the fitted ARIMARegressor. |
| *load* | Loads component at file path. |
| *needs_fitting* | Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances. |
| *parameters* | Returns the parameters which were used to initialize the component. |
| *predict* | Make predictions using fitted ARIMA regressor. |
| *predict_proba* | Make probability estimates for labels. |
| *save* | Saves component at file path. |
| *update_parameters* | Updates the parameter dictionary of the component. |

**clone**(*self*)

Constructs a new component with the same parameters and random state.

**Returns** A new instance of this component with identical parameters and random state.

**default_parameters**(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

**Returns** Default parameters for this component.

**Return type** dict

**describe**(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

**Parameters**

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description as dictionary in the format {"name": name, "parameters": parameters}

**Returns** Returns dictionary if return_dict is True, else None.

**Return type** None or dict

**property feature_importance**(*self*) → numpy.ndarray

Returns array of 0's with a length of 1 as feature_importance is not defined for ARIMA regressor.

**fit**(*self, X: pandas.DataFrame, y: Optional[pandas.Series] = None*)

Fits ARIMA regressor to data.

**Parameters**

- **X** (*pd.DataFrame*) – The input training data of shape [n_samples, n_features].
- **y** (*pd.Series*) – The target training data of length [n_samples].

**Returns** self

**Raises** **ValueError** – If y was not passed in.

**get_prediction_intervals**(*self, X: pandas.DataFrame, y: pandas.Series = None, coverage: List[float] = None, predictions: pandas.Series = None*) → Dict[str, pandas.Series]

Find the prediction intervals using the fitted ARIMARegressor.

**Parameters**

- **X** (*pd.DataFrame*) – Data of shape [n_samples, n_features].
- **y** (*pd.Series*) – Target data. Optional.
- **coverage** (*list[float]*) – A list of floats between the values 0 and 1 that the upper and lower bounds of the prediction interval should be calculated for.
- **predictions** (*pd.Series*) – Not used for ARIMA regressor.

**Returns** Prediction intervals, keys are in the format {coverage}_lower or {coverage}_upper.

**Return type** dict

**static load**(*file_path*)

Loads component at file path.

**Parameters** **file_path** (*str*) – Location to load file.

**Returns** ComponentBase object

**needs_fitting**(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need to be fit or whose fit methods do nothing.

**Returns** True.

**property parameters**(*self*)

Returns the parameters which were used to initialize the component.

**predict**(*self*, *X: pandas.DataFrame*, *y: Optional[pandas.Series] = None*) → pandas.Series

Make predictions using fitted ARIMA regressor.

**Parameters**

- **X** (*pd.DataFrame*) – Data of shape [n_samples, n_features].
- **y** (*pd.Series*) – Target data.

**Returns** Predicted values.

**Return type** pd.Series

**Raises** **ValueError** – If X was passed to *fit* but not passed in *predict*.

**predict_proba**(*self*, *X: pandas.DataFrame*) → pandas.Series

Make probability estimates for labels.

**Parameters** **X** (*pd.DataFrame*) – Features.

**Returns** Probability estimates.

**Return type** pd.Series

**Raises** **MethodPropertyNotFoundError** – If estimator does not have a predict_proba method or a component_obj that implements predict_proba.

**save**(*self*, *file_path*, *pickle_protocol=cloudpickle.DEFAULT_PROTOCOL*)

Saves component at file path.

**Parameters**

- **file_path** (*str*) – Location to save file.
- **pickle_protocol** (*int*) – The pickle data stream format.

**update_parameters**(*self*, *update_dict*, *reset_fit=True*)

Updates the parameter dictionary of the component.

**Parameters**

- **update_dict** (*dict*) – A dict of parameters to update.
- **reset_fit** (*bool, optional*) – If True, will set *_is_fitted* to False.

**class** evalml.pipelines.components.estimators.regressors.**BaselineRegressor**(*strategy='mean'*, *random_seed=0*, ***kwargs*)

Baseline regressor that uses a simple strategy to make predictions. This is useful as a simple baseline regressor to compare with other regressors.

**Parameters**

- **strategy** (*str*) – Method used to predict. Valid options are "mean", "median". Defaults to "mean".
- **random_seed** (*int*) – Seed for the random number generator. Defaults to 0.

**Attributes**

| | |
|---|---|
| **hyper-parameter_ranges** | {} |
| **model_family** | ModelFamily.BASELINE |
| **modifies_features** | True |
| **modifies_target** | False |
| **name** | Baseline Regressor |
| **supported_problem_types** | [ ProblemTypes.REGRESSION, ProblemTypes.TIME_SERIES_REGRESSION,] |
| **training_only** | False |

**Methods**

| | |
|---|---|
| *clone* | Constructs a new component with the same parameters and random state. |
| *default_parameters* | Returns the default parameters for this component. |
| *describe* | Describe a component and its parameters. |
| *feature_importance* | Returns importance associated with each feature. Since baseline regressors do not use input features to calculate predictions, returns an array of zeroes. |
| *fit* | Fits baseline regression component to data. |
| *get_prediction_intervals* | Find the prediction intervals using the fitted regressor. |
| *load* | Loads component at file path. |
| *needs_fitting* | Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances. |
| *parameters* | Returns the parameters which were used to initialize the component. |
| *predict* | Make predictions using the baseline regression strategy. |
| *predict_proba* | Make probability estimates for labels. |
| *save* | Saves component at file path. |
| *update_parameters* | Updates the parameter dictionary of the component. |

**clone**(*self*)

Constructs a new component with the same parameters and random state.

**Returns** A new instance of this component with identical parameters and random state.

**default_parameters**(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

**Returns** Default parameters for this component.

**Return type** dict

**describe**(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

**Parameters**

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description as dictionary in the format {"name": name, "parameters": parameters}

**Returns** Returns dictionary if return_dict is True, else None.

**Return type** None or dict

**property feature_importance**(*self*)

Returns importance associated with each feature. Since baseline regressors do not use input features to calculate predictions, returns an array of zeroes.

**Returns** An array of zeroes.

**Return type** np.ndarray (float)

**fit**(*self*, *X*, *y=None*)

Fits baseline regression component to data.

**Parameters**

- **X** (*pd.DataFrame*) – The input training data of shape [n_samples, n_features].
- **y** (*pd.Series*) – The target training data of length [n_samples].

**Returns** self

**Raises** **ValueError** – If input y is None.

**get_prediction_intervals**(*self*, *X: pandas.DataFrame*, *y: Optional[pandas.Series] = None*, *coverage: List[float] = None*, *predictions: pandas.Series = None*) → Dict[str, pandas.Series]

Find the prediction intervals using the fitted regressor.

This function takes the predictions of the fitted estimator and calculates the rolling standard deviation across all predictions using a window size of 5. The lower and upper predictions are determined by taking the percent point (quantile) function of the lower tail probability at each bound multiplied by the rolling standard deviation.

**Parameters**

- **X** (*pd.DataFrame*) – Data of shape [n_samples, n_features].
- **y** (*pd.Series*) – Target data. Ignored.
- **coverage** (*list[float]*) – A list of floats between the values 0 and 1 that the upper and lower bounds of the prediction interval should be calculated for.
- **predictions** (*pd.Series*) – Optional list of predictions to use. If None, will generate predictions using *X*.

**Returns** Prediction intervals, keys are in the format {coverage}_lower or {coverage}_upper.

**Return type** dict

**Raises** **MethodPropertyNotFoundError** – If the estimator does not support Time Series Regression as a problem type.

**static load**(*file_path*)

Loads component at file path.

**Parameters** **file_path** (*str*) – Location to load file.

**Returns** ComponentBase object

**needs_fitting**(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need to be fit or whose fit methods do nothing.

**Returns** True.

**property parameters**(*self*)

Returns the parameters which were used to initialize the component.

**predict**(*self*, *X*)

Make predictions using the baseline regression strategy.

**Parameters** **X** (*pd.DataFrame*) – Data of shape [n_samples, n_features].

**Returns** Predicted values.

**Return type** pd.Series

**predict_proba**(*self*, *X: pandas.DataFrame*) → pandas.Series

Make probability estimates for labels.

**Parameters** **X** (*pd.DataFrame*) – Features.

**Returns** Probability estimates.

**Return type** pd.Series

**Raises** **MethodPropertyNotFoundError** – If estimator does not have a predict_proba method or a component_obj that implements predict_proba.

**save**(*self*, *file_path*, *pickle_protocol=cloudpickle.DEFAULT_PROTOCOL*)

Saves component at file path.

**Parameters**

- **file_path** (*str*) – Location to save file.
- **pickle_protocol** (*int*) – The pickle data stream format.

**update_parameters**(*self*, *update_dict*, *reset_fit=True*)

Updates the parameter dictionary of the component.

**Parameters**

- **update_dict** (*dict*) – A dict of parameters to update.
- **reset_fit** (*bool, optional*) – If True, will set *_is_fitted* to False.

**class** evalml.pipelines.components.estimators.regressors.**CatBoostRegressor**(*n_estimators=10*, *eta=0.03*, *max_depth=6*, *bootstrap_type=None*, *silent=False*, *allow_writing_files=False*, *random_seed=0*, *n_jobs=- 1*, ***kwargs*)

CatBoost Regressor, a regressor that uses gradient-boosting on decision trees. CatBoost is an open-source library and natively supports categorical features.

For more information, check out https://catboost.ai/

**Parameters**

- **n_estimators** (*float*) – The maximum number of trees to build. Defaults to 10.
- **eta** (*float*) – The learning rate. Defaults to 0.03.
- **max_depth** (*int*) – The maximum tree depth for base learners. Defaults to 6.
- **bootstrap_type** (*string*) – Defines the method for sampling the weights of objects. Available methods are 'Bayesian', 'Bernoulli', 'MVS'. Defaults to None.
- **silent** (*boolean*) – Whether to use the "silent" logging mode. Defaults to True.
- **allow_writing_files** (*boolean*) – Whether to allow writing snapshot files while training. Defaults to False.
- **n_jobs** (*int or None*) – Number of jobs to run in parallel. -1 uses all processes. Defaults to -1.
- **random_seed** (*int*) – Seed for the random number generator. Defaults to 0.

**Attributes**

| | |
|---|---|
| **hyper-parameter_ranges** | { "n_estimators": Integer(4, 100), "eta": Real(0.000001, 1), "max_depth": Integer(4, 10),} |
| **model_family** | ModelFamily.CATBOOST |
| **modifies_features** | True |
| **modifies_target** | False |
| **name** | CatBoost Regressor |
| **supported_problem_types** | [ ProblemTypes.REGRESSION, ProblemTypes.TIME_SERIES_REGRESSION,] |
| **training_only** | False |

**Methods**

| | |
|---|---|
| *clone* | Constructs a new component with the same parameters and random state. |
| *default_parameters* | Returns the default parameters for this component. |
| *describe* | Describe a component and its parameters. |
| *feature_importance* | Feature importance of fitted CatBoost regressor. |
| *fit* | Fits CatBoost regressor component to data. |
| *get_prediction_intervals* | Find the prediction intervals using the fitted regressor. |
| *load* | Loads component at file path. |
| *needs_fitting* | Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances. |
| *parameters* | Returns the parameters which were used to initialize the component. |
| *predict* | Make predictions using the fitted CatBoost regressor. |
| *predict_proba* | Make probability estimates for labels. |
| *save* | Saves component at file path. |
| *update_parameters* | Updates the parameter dictionary of the component. |

**clone**(*self*)

Constructs a new component with the same parameters and random state.

**Returns** A new instance of this component with identical parameters and random state.

**default_parameters**(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

**Returns** Default parameters for this component.

**Return type** dict

**describe**(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

**Parameters**

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description as dictionary in the format {"name": name, "parameters": parameters}

**Returns** Returns dictionary if return_dict is True, else None.

**Return type** None or dict

**property feature_importance**(*self*)

Feature importance of fitted CatBoost regressor.

**fit**(*self*, *X*, *y=None*)

Fits CatBoost regressor component to data.

**Parameters**

- **X** (*pd.DataFrame*) – The input training data of shape [n_samples, n_features].
- **y** (*pd.Series*) – The target training data of length [n_samples].

**Returns** self

**get_prediction_intervals**(*self*, *X: pandas.DataFrame*, *y: Optional[pandas.Series] = None*, *coverage: List[float] = None*, *predictions: pandas.Series = None*) → Dict[str, pandas.Series]

Find the prediction intervals using the fitted regressor.

This function takes the predictions of the fitted estimator and calculates the rolling standard deviation across all predictions using a window size of 5. The lower and upper predictions are determined by taking the percent point (quantile) function of the lower tail probability at each bound multiplied by the rolling standard deviation.

**Parameters**

- **X** (*pd.DataFrame*) – Data of shape [n_samples, n_features].
- **y** (*pd.Series*) – Target data. Ignored.
- **coverage** (*list[float]*) – A list of floats between the values 0 and 1 that the upper and lower bounds of the prediction interval should be calculated for.
- **predictions** (*pd.Series*) – Optional list of predictions to use. If None, will generate predictions using *X*.

**Returns** Prediction intervals, keys are in the format {coverage}_lower or {coverage}_upper.

**Return type** dict

**Raises** **MethodPropertyNotFoundError** – If the estimator does not support Time Series Regression as a problem type.

**static load**(*file_path*)

Loads component at file path.

**Parameters** **file_path** (*str*) – Location to load file.

**Returns** ComponentBase object

**needs_fitting**(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need to be fit or whose fit methods do nothing.

**Returns** True.

**property parameters**(*self*)

Returns the parameters which were used to initialize the component.

**predict**(*self*, *X*)

Make predictions using the fitted CatBoost regressor.

**Parameters** **X** (*pd.DataFrame*) – Data of shape [n_samples, n_features].

**Returns** Predicted values.

**Return type** pd.DataFrame

**predict_proba**(*self*, *X: pandas.DataFrame*) → pandas.Series

Make probability estimates for labels.

**Parameters** **X** (*pd.DataFrame*) – Features.

**Returns** Probability estimates.

**Return type** pd.Series

**Raises** **MethodPropertyNotFoundError** – If estimator does not have a predict_proba method or a component_obj that implements predict_proba.

**save**(*self*, *file_path*, *pickle_protocol=cloudpickle.DEFAULT_PROTOCOL*)

Saves component at file path.

**Parameters**

- **file_path** (*str*) – Location to save file.
- **pickle_protocol** (*int*) – The pickle data stream format.

**update_parameters**(*self*, *update_dict*, *reset_fit=True*)

Updates the parameter dictionary of the component.

**Parameters**

- **update_dict** (*dict*) – A dict of parameters to update.
- **reset_fit** (*bool, optional*) – If True, will set *_is_fitted* to False.

**class** evalml.pipelines.components.estimators.regressors.**DecisionTreeRegressor**(*criterion='squared_error', max_features='auto', max_depth=6, min_samples_split=2, min_weight_fraction_leaf=0.0, random_seed=0, **kwargs*)

Decision Tree Regressor.

**Parameters**

- **criterion** (*{"squared_error", "friedman_mse", "absolute_error", "poisson"}*) – The function to measure the quality of a split. Supported criteria are:
  - "squared_error" for the mean squared error, which is equal to variance reduction as feature selection criterion and minimizes the L2 loss using the mean of each terminal node
  - "friedman_mse", which uses mean squared error with Friedman"s improvement score for potential splits
  - "absolute_error" for the mean absolute error, which minimizes the L1 loss using the median of each terminal node,
  - "poisson" which uses reduction in Poisson deviance to find splits.
- **max_features** (*int, float or {"auto", "sqrt", "log2"}*) – The number of features to consider when looking for the best split:
  - If int, then consider max_features features at each split.
  - If float, then max_features is a fraction and int(max_features * n_features) features are considered at each split.
  - If "auto", then max_features=sqrt(n_features).
  - If "sqrt", then max_features=sqrt(n_features).
  - If "log2", then max_features=log2(n_features).
  - If None, then max_features = n_features.

The search for a split does not stop until at least one valid partition of the node samples is found, even if it requires to effectively inspect more than max_features features.

- **max_depth** (*int*) – The maximum depth of the tree. Defaults to 6.
- **min_samples_split** (*int or float*) – The minimum number of samples required to split an internal node:
  - If int, then consider min_samples_split as the minimum number.
  - If float, then min_samples_split is a fraction and ceil(min_samples_split * n_samples) are the minimum number of samples for each split.

  Defaults to 2.
- **min_weight_fraction_leaf** (*float*) – The minimum weighted fraction of the sum total of weights (of all the input samples) required to be at a leaf node. Defaults to 0.0.
- **random_seed** (*int*) – Seed for the random number generator. Defaults to 0.

**Attributes**

| | |
|---|---|
| **hyper-parameter_ranges** | { "criterion": ["squared_error", "friedman_mse", "absolute_error"], "max_features": ["auto", "sqrt", "log2"], "max_depth": Integer(4, 10),} |
| **model_family** | ModelFamily.DECISION_TREE |
| **modifies_features** | True |
| **modifies_target** | False |
| **name** | Decision Tree Regressor |
| **supported_problem_types** | [ ProblemTypes.REGRESSION, ProblemTypes.TIME_SERIES_REGRESSION,] |
| **training_only** | False |

**Methods**

| | |
|---|---|
| *clone* | Constructs a new component with the same parameters and random state. |
| *default_parameters* | Returns the default parameters for this component. |
| *describe* | Describe a component and its parameters. |
| *feature_importance* | Returns importance associated with each feature. |
| *fit* | Fits estimator to data. |
| *get_prediction_intervals* | Find the prediction intervals using the fitted regressor. |
| *load* | Loads component at file path. |
| *needs_fitting* | Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances. |
| *parameters* | Returns the parameters which were used to initialize the component. |
| *predict* | Make predictions using selected features. |
| *predict_proba* | Make probability estimates for labels. |
| *save* | Saves component at file path. |
| *update_parameters* | Updates the parameter dictionary of the component. |

**clone**(*self*)

Constructs a new component with the same parameters and random state.

**Returns** A new instance of this component with identical parameters and random state.

**default_parameters**(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

**Returns** Default parameters for this component.

**Return type** dict

**describe**(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

**Parameters**

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description as dictionary in the format {"name": name, "parameters": parameters}

**Returns** Returns dictionary if return_dict is True, else None.

**Return type** None or dict

**property feature_importance**(*self*) → pandas.Series

Returns importance associated with each feature.

**Returns** Importance associated with each feature.

**Return type** np.ndarray

**Raises** **MethodPropertyNotFoundError** – If estimator does not have a feature_importance method or a component_obj that implements feature_importance.

**fit**(*self*, *X: pandas.DataFrame*, *y: Optional[pandas.Series] = None*)

Fits estimator to data.

**Parameters**

- **X** (*pd.DataFrame*) – The input training data of shape [n_samples, n_features].
- **y** (*pd.Series, optional*) – The target training data of length [n_samples].

**Returns** self

**get_prediction_intervals**(*self*, *X: pandas.DataFrame*, *y: Optional[pandas.Series] = None*, *coverage: List[float] = None*, *predictions: pandas.Series = None*) → Dict[str, pandas.Series]

Find the prediction intervals using the fitted regressor.

This function takes the predictions of the fitted estimator and calculates the rolling standard deviation across all predictions using a window size of 5. The lower and upper predictions are determined by taking the percent point (quantile) function of the lower tail probability at each bound multiplied by the rolling standard deviation.

**Parameters**

- **X** (*pd.DataFrame*) – Data of shape [n_samples, n_features].
- **y** (*pd.Series*) – Target data. Ignored.

- **coverage** (*list[float]*) – A list of floats between the values 0 and 1 that the upper and lower bounds of the prediction interval should be calculated for.
- **predictions** (*pd.Series*) – Optional list of predictions to use. If None, will generate predictions using *X*.

**Returns** Prediction intervals, keys are in the format {coverage}_lower or {coverage}_upper.

**Return type** dict

**Raises** **MethodPropertyNotFoundError** – If the estimator does not support Time Series Regression as a problem type.

**static load**(*file_path*)

Loads component at file path.

**Parameters** **file_path** (*str*) – Location to load file.

**Returns** ComponentBase object

**needs_fitting**(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need to be fit or whose fit methods do nothing.

**Returns** True.

**property parameters**(*self*)

Returns the parameters which were used to initialize the component.

**predict**(*self*, *X: pandas.DataFrame*) → pandas.Series

Make predictions using selected features.

**Parameters** **X** (*pd.DataFrame*) – Data of shape [n_samples, n_features].

**Returns** Predicted values.

**Return type** pd.Series

**Raises** **MethodPropertyNotFoundError** – If estimator does not have a predict method or a component_obj that implements predict.

**predict_proba**(*self*, *X: pandas.DataFrame*) → pandas.Series

Make probability estimates for labels.

**Parameters** **X** (*pd.DataFrame*) – Features.

**Returns** Probability estimates.

**Return type** pd.Series

**Raises** **MethodPropertyNotFoundError** – If estimator does not have a predict_proba method or a component_obj that implements predict_proba.

**save**(*self*, *file_path*, *pickle_protocol=cloudpickle.DEFAULT_PROTOCOL*)

Saves component at file path.

**Parameters**

- **file_path** (*str*) – Location to save file.
- **pickle_protocol** (*int*) – The pickle data stream format.

**update_parameters**(*self*, *update_dict*, *reset_fit=True*)

Updates the parameter dictionary of the component.

**Parameters**

- **update_dict** (*dict*) – A dict of parameters to update.
- **reset_fit** (*bool, optional*) – If True, will set *_is_fitted* to False.

**class** evalml.pipelines.components.estimators.regressors.**ElasticNetRegressor**(*alpha=0.0001*, *l1_ratio=0.15*, *max_iter=1000*, *random_seed=0*, ***kwargs*)

Elastic Net Regressor.

**Parameters**

- **alpha** (*float*) – Constant that multiplies the penalty terms. Defaults to 0.0001.
- **l1_ratio** (*float*) – The mixing parameter, with 0 <= l1_ratio <= 1. Only used if penalty='elasticnet'. Setting l1_ratio=0 is equivalent to using penalty='l2', while setting l1_ratio=1 is equivalent to using penalty='l1'. For 0 < l1_ratio <1, the penalty is a combination of L1 and L2. Defaults to 0.15.
- **max_iter** (*int*) – The maximum number of iterations. Defaults to 1000.
- **random_seed** (*int*) – Seed for the random number generator. Defaults to 0.

**Attributes**

| | |
|---|---|
| **hyper-parameter_ranges** | { "alpha": Real(0, 1), "l1_ratio": Real(0, 1),} |
| **model_family** | ModelFamily.LINEAR_MODEL |
| **modifies_features** | True |
| **modifies_target** | False |
| **name** | Elastic Net Regressor |
| **supported_problem_types** | [ ProblemTypes.REGRESSION, ProblemTypes.TIME_SERIES_REGRESSION,] |
| **training_only** | False |

**Methods**

| | |
|---|---|
| *clone* | Constructs a new component with the same parameters and random state. |
| *default_parameters* | Returns the default parameters for this component. |
| *describe* | Describe a component and its parameters. |
| *feature_importance* | Feature importance for fitted ElasticNet regressor. |
| *fit* | Fits estimator to data. |
| *get_prediction_intervals* | Find the prediction intervals using the fitted regressor. |
| *load* | Loads component at file path. |
| *needs_fitting* | Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances. |
| *parameters* | Returns the parameters which were used to initialize the component. |
| *predict* | Make predictions using selected features. |
| *predict_proba* | Make probability estimates for labels. |
| *save* | Saves component at file path. |
| *update_parameters* | Updates the parameter dictionary of the component. |

**clone**(*self*)

Constructs a new component with the same parameters and random state.

**Returns** A new instance of this component with identical parameters and random state.

**default_parameters**(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

**Returns** Default parameters for this component.

**Return type** dict

**describe**(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

**Parameters**

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description as dictionary in the format {"name": name, "parameters": parameters}

**Returns** Returns dictionary if return_dict is True, else None.

**Return type** None or dict

**property feature_importance**(*self*)

Feature importance for fitted ElasticNet regressor.

**fit**(*self*, *X: pandas.DataFrame*, *y: Optional[pandas.Series] = None*)

Fits estimator to data.

**Parameters**

- **X** (*pd.DataFrame*) – The input training data of shape [n_samples, n_features].
- **y** (*pd.Series, optional*) – The target training data of length [n_samples].

**Returns** self

**get_prediction_intervals**(*self*, *X: pandas.DataFrame*, *y: Optional[pandas.Series] = None*, *coverage: List[float] = None*, *predictions: pandas.Series = None*) → Dict[str, pandas.Series]

Find the prediction intervals using the fitted regressor.

This function takes the predictions of the fitted estimator and calculates the rolling standard deviation across all predictions using a window size of 5. The lower and upper predictions are determined by taking the percent point (quantile) function of the lower tail probability at each bound multiplied by the rolling standard deviation.

**Parameters**

- **X** (*pd.DataFrame*) – Data of shape [n_samples, n_features].
- **y** (*pd.Series*) – Target data. Ignored.
- **coverage** (*list[float]*) – A list of floats between the values 0 and 1 that the upper and lower bounds of the prediction interval should be calculated for.
- **predictions** (*pd.Series*) – Optional list of predictions to use. If None, will generate predictions using *X*.

**Returns** Prediction intervals, keys are in the format {coverage}_lower or {coverage}_upper.

**Return type** dict

**Raises** **MethodPropertyNotFoundError** – If the estimator does not support Time Series Regression as a problem type.

**static load**(*file_path*)

Loads component at file path.

**Parameters** **file_path** (*str*) – Location to load file.

**Returns** ComponentBase object

**needs_fitting**(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need to be fit or whose fit methods do nothing.

**Returns** True.

**property parameters**(*self*)

Returns the parameters which were used to initialize the component.

**predict**(*self*, *X: pandas.DataFrame*) → pandas.Series

Make predictions using selected features.

**Parameters** **X** (*pd.DataFrame*) – Data of shape [n_samples, n_features].

**Returns** Predicted values.

**Return type** pd.Series

**Raises** **MethodPropertyNotFoundError** – If estimator does not have a predict method or a component_obj that implements predict.

**predict_proba**(*self*, *X: pandas.DataFrame*) → pandas.Series

Make probability estimates for labels.

**Parameters** **X** (*pd.DataFrame*) – Features.

**Returns** Probability estimates.

**Return type** pd.Series

**Raises** **MethodPropertyNotFoundError** – If estimator does not have a predict_proba method or a component_obj that implements predict_proba.

**save**(*self*, *file_path*, *pickle_protocol=cloudpickle.DEFAULT_PROTOCOL*)

Saves component at file path.

**Parameters**

- **file_path** (*str*) – Location to save file.
- **pickle_protocol** (*int*) – The pickle data stream format.

**update_parameters**(*self*, *update_dict*, *reset_fit=True*)

Updates the parameter dictionary of the component.

**Parameters**

- **update_dict** (*dict*) – A dict of parameters to update.
- **reset_fit** (*bool, optional*) – If True, will set *_is_fitted* to False.

class evalml.pipelines.components.estimators.regressors.**ExponentialSmoothingRegressor**(*trend: Optional[str] = None, damped_trend: bool = False, seasonal: Optional[str] = None, sp: int = 2, n_jobs: int = -1, random_seed: Union[int, float] = 0, **kwargs*)

Holt-Winters Exponential Smoothing Forecaster.

Currently ExponentialSmoothingRegressor isn't supported via conda install. It's recommended that it be installed via PyPI.

**Parameters**

- **trend** (*str*) – Type of trend component. Defaults to None.
- **damped_trend** (*bool*) – If the trend component should be damped. Defaults to False.
- **seasonal** (*str*) – Type of seasonal component. Takes one of {"additive", None}. Can also be multiplicative if
- **0** (*none of the target data is*) –
- **None.** (*but AutoMLSearch wiill not tune for this. Defaults to*) –
- **sp** (*int*) – The number of seasonal periods to consider. Defaults to 2.
- **n_jobs** (*int or None*) – Non-negative integer describing level of parallelism used for pipelines. Defaults to -1.
- **random_seed** (*int*) – Seed for the random number generator. Defaults to 0.

**Attributes**

| | |
|---|---|
| **hyper-parameter_ranges** | { "trend": [None, "additive"], "damped_trend": [True, False], "seasonal": [None, "additive"], "sp": Integer(2, 8),} |
| **model_family** | ModelFamily.EXPONENTIAL_SMOOTHING |
| **modifies_features** | True |
| **modifies_target** | False |
| **name** | Exponential Smoothing Regressor |
| **supported_problem_types** | [ProblemTypes.TIME_SERIES_REGRESSION] |
| **training_only** | False |

**Methods**

| | |
|---|---|
| *clone* | Constructs a new component with the same parameters and random state. |
| *default_parameters* | Returns the default parameters for this component. |
| *describe* | Describe a component and its parameters. |
| *feature_importance* | Returns array of 0's with a length of 1 as feature_importance is not defined for Exponential Smoothing regressor. |
| *fit* | Fits Exponential Smoothing Regressor to data. |
| *get_prediction_intervals* | Find the prediction intervals using the fitted ExponentialSmoothingRegressor. |
| *load* | Loads component at file path. |
| *needs_fitting* | Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances. |
| *parameters* | Returns the parameters which were used to initialize the component. |
| *predict* | Make predictions using fitted Exponential Smoothing regressor. |
| *predict_proba* | Make probability estimates for labels. |
| *save* | Saves component at file path. |
| *update_parameters* | Updates the parameter dictionary of the component. |

**clone**(*self*)

Constructs a new component with the same parameters and random state.

**Returns** A new instance of this component with identical parameters and random state.

**default_parameters**(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

**Returns** Default parameters for this component.

**Return type** dict

**describe**(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

**Parameters**

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description as dictionary in the format {"name": name, "parameters": parameters}

**Returns** Returns dictionary if return_dict is True, else None.

**Return type** None or dict

**property feature_importance**(*self*) → pandas.Series

Returns array of 0's with a length of 1 as feature_importance is not defined for Exponential Smoothing regressor.

**fit**(*self*, *X: pandas.DataFrame*, *y: Optional[pandas.Series] = None*)

Fits Exponential Smoothing Regressor to data.

**Parameters**

- **X** (*pd.DataFrame*) – The input training data of shape [n_samples, n_features]. Ignored.
- **y** (*pd.Series*) – The target training data of length [n_samples].

**Returns** self

**Raises** **ValueError** – If y was not passed in.

**get_prediction_intervals**(*self*, *X: pandas.DataFrame*, *y: Optional[pandas.Series] = None*, *coverage: List[float] = None*, *predictions: pandas.Series = None*) → Dict[str, pandas.Series]

Find the prediction intervals using the fitted ExponentialSmoothingRegressor.

Calculates the prediction intervals by using a simulation of the time series following a specified state space model.

**Parameters**

- **X** (*pd.DataFrame*) – Data of shape [n_samples, n_features].
- **y** (*pd.Series*) – Target data. Optional.
- **coverage** (*List[float]*) – A list of floats between the values 0 and 1 that the upper and lower bounds of the prediction interval should be calculated for.
- **predictions** (*pd.Series*) – Not used for Exponential Smoothing regressor.

**Returns** Prediction intervals, keys are in the format {coverage}_lower or {coverage}_upper.

**Return type** dict

**static load**(*file_path*)

Loads component at file path.

**Parameters** **file_path** (*str*) – Location to load file.

**Returns** ComponentBase object

**needs_fitting**(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need to be fit or whose fit methods do nothing.

**Returns** True.

**property parameters**(*self*)

Returns the parameters which were used to initialize the component.

**predict**(*self*, *X: pandas.DataFrame*, *y: Optional[pandas.Series] = None*) → pandas.Series

Make predictions using fitted Exponential Smoothing regressor.

**Parameters**

- **X** (*pd.DataFrame*) – Data of shape [n_samples, n_features]. Ignored except to set forecast horizon.
- **y** (*pd.Series*) – Target data.

**Returns** Predicted values.

**Return type** pd.Series

**predict_proba**(*self*, *X: pandas.DataFrame*) → pandas.Series

Make probability estimates for labels.

**Parameters** **X** (*pd.DataFrame*) – Features.

**Returns** Probability estimates.

**Return type** pd.Series

**Raises** **MethodPropertyNotFoundError** – If estimator does not have a predict_proba method or a component_obj that implements predict_proba.

**save**(*self*, *file_path*, *pickle_protocol=cloudpickle.DEFAULT_PROTOCOL*)

Saves component at file path.

**Parameters**

- **file_path** (*str*) – Location to save file.
- **pickle_protocol** (*int*) – The pickle data stream format.

**update_parameters**(*self*, *update_dict*, *reset_fit=True*)

Updates the parameter dictionary of the component.

**Parameters**

- **update_dict** (*dict*) – A dict of parameters to update.
- **reset_fit** (*bool, optional*) – If True, will set *_is_fitted* to False.

**class** evalml.pipelines.components.estimators.regressors.**ExtraTreesRegressor**(*n_estimators: int = 100*, *max_features: str = 'auto'*, *max_depth: int = 6*, *min_samples_split: int = 2*, *min_weight_fraction_leaf: float = 0.0*, *n_jobs: int = - 1*, *random_seed: Union[int, float] = 0*, ***kwargs*)

Extra Trees Regressor.

**Parameters**

- **n_estimators** (*float*) – The number of trees in the forest. Defaults to 100.
- **max_features** (*int, float or {"auto", "sqrt", "log2"}*) – The number of features to consider when looking for the best split:
  - If int, then consider max_features features at each split.
  - If float, then max_features is a fraction and int(max_features * n_features) features are considered at each split.
  - If "auto", then max_features=sqrt(n_features).
  - If "sqrt", then max_features=sqrt(n_features).
  - If "log2", then max_features=log2(n_features).
  - If None, then max_features = n_features.

  The search for a split does not stop until at least one valid partition of the node samples is found, even if it requires to effectively inspect more than max_features features. Defaults to "auto".
- **max_depth** (*int*) – The maximum depth of the tree. Defaults to 6.
- **min_samples_split** (*int or float*) – The minimum number of samples required to split an internal node:
  - If int, then consider min_samples_split as the minimum number.
  - If float, then min_samples_split is a fraction and ceil(min_samples_split * n_samples) are the minimum number of samples for each split.
- **2.** (*Defaults to*) –
- **min_weight_fraction_leaf** (*float*) – The minimum weighted fraction of the sum total of weights (of all the input samples) required to be at a leaf node. Defaults to 0.0.
- **n_jobs** (*int or None*) – Number of jobs to run in parallel. -1 uses all processes. Defaults to -1.
- **random_seed** (*int*) – Seed for the random number generator. Defaults to 0.

**Attributes**

| | |
|---|---|
| **hyper-parameter_ranges** | { "n_estimators": Integer(10, 1000), "max_features": ["auto", "sqrt", "log2"], "max_depth": Integer(4, 10),} |
| **model_family** | ModelFamily.EXTRA_TREES |
| **modifies_features** | True |
| **modifies_target** | False |
| **name** | Extra Trees Regressor |
| **supported_problem_types** | [ ProblemTypes.REGRESSION, ProblemTypes.TIME_SERIES_REGRESSION,] |
| **training_only** | False |

**Methods**

| | |
|---|---|
| *clone* | Constructs a new component with the same parameters and random state. |
| *default_parameters* | Returns the default parameters for this component. |
| *describe* | Describe a component and its parameters. |
| *feature_importance* | Returns importance associated with each feature. |
| *fit* | Fits estimator to data. |
| *get_prediction_intervals* | Find the prediction intervals using the fitted ExtraTreesRegressor. |
| *load* | Loads component at file path. |
| *needs_fitting* | Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances. |
| *parameters* | Returns the parameters which were used to initialize the component. |
| *predict* | Make predictions using selected features. |
| *predict_proba* | Make probability estimates for labels. |
| *save* | Saves component at file path. |
| *update_parameters* | Updates the parameter dictionary of the component. |

**clone**(*self*)

Constructs a new component with the same parameters and random state.

**Returns** A new instance of this component with identical parameters and random state.

**default_parameters**(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

**Returns** Default parameters for this component.

**Return type** dict

**describe**(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

**Parameters**

- **print_name** (*bool, optional*) – whether to print name of component

- **return_dict** (*bool, optional*) – whether to return description as dictionary in the format {"name": name, "parameters": parameters}

**Returns** Returns dictionary if return_dict is True, else None.

**Return type** None or dict

**property feature_importance**(*self*) → pandas.Series

Returns importance associated with each feature.

**Returns** Importance associated with each feature.

**Return type** np.ndarray

**Raises MethodPropertyNotFoundError** – If estimator does not have a feature_importance method or a component_obj that implements feature_importance.

**fit**(*self, X: pandas.DataFrame, y: Optional[pandas.Series] = None*)

Fits estimator to data.

**Parameters**

- **X** (*pd.DataFrame*) – The input training data of shape [n_samples, n_features].
- **y** (*pd.Series, optional*) – The target training data of length [n_samples].

**Returns** self

**get_prediction_intervals**(*self, X: pandas.DataFrame, y: Optional[pandas.Series] = None, coverage: List[float] = None, predictions: pandas.Series = None*) → Dict[str, pandas.Series]

Find the prediction intervals using the fitted ExtraTreesRegressor.

**Parameters**

- **X** (*pd.DataFrame*) – Data of shape [n_samples, n_features].
- **y** (*pd.Series*) – Target data. Optional.
- **coverage** (*list[float]*) – A list of floats between the values 0 and 1 that the upper and lower bounds of the prediction interval should be calculated for.
- **predictions** (*pd.Series*) – Optional list of predictions to use. If None, will generate predictions using *X*.

**Returns** Prediction intervals, keys are in the format {coverage}_lower or {coverage}_upper.

**Return type** dict

**static load**(*file_path*)

Loads component at file path.

**Parameters file_path** (*str*) – Location to load file.

**Returns** ComponentBase object

**needs_fitting**(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need to be fit or whose fit methods do nothing.

**Returns** True.

**property parameters**(*self*)

Returns the parameters which were used to initialize the component.

**predict**(*self*, *X: pandas.DataFrame*) → pandas.Series

Make predictions using selected features.

**Parameters** **X** (*pd.DataFrame*) – Data of shape [n_samples, n_features].

**Returns** Predicted values.

**Return type** pd.Series

**Raises** **MethodPropertyNotFoundError** – If estimator does not have a predict method or a component_obj that implements predict.

**predict_proba**(*self*, *X: pandas.DataFrame*) → pandas.Series

Make probability estimates for labels.

**Parameters** **X** (*pd.DataFrame*) – Features.

**Returns** Probability estimates.

**Return type** pd.Series

**Raises** **MethodPropertyNotFoundError** – If estimator does not have a predict_proba method or a component_obj that implements predict_proba.

**save**(*self*, *file_path*, *pickle_protocol=cloudpickle.DEFAULT_PROTOCOL*)

Saves component at file path.

**Parameters**

- **file_path** (*str*) – Location to save file.
- **pickle_protocol** (*int*) – The pickle data stream format.

**update_parameters**(*self*, *update_dict*, *reset_fit=True*)

Updates the parameter dictionary of the component.

**Parameters**

- **update_dict** (*dict*) – A dict of parameters to update.
- **reset_fit** (*bool, optional*) – If True, will set *_is_fitted* to False.

**class** evalml.pipelines.components.estimators.regressors.**LightGBMRegressor**(*boosting_type='gbdt', learning_rate=0.1, n_estimators=20, max_depth=0, num_leaves=31, min_child_samples=20, bagging_fraction=0.9, bagging_freq=0, n_jobs=- 1, random_seed=0, **kwargs*)

LightGBM Regressor.

**Parameters**

- **boosting_type** (*string*) – Type of boosting to use. Defaults to "gbdt". - 'gbdt' uses traditional Gradient Boosting Decision Tree - "dart", uses Dropouts meet Multiple Additive Regression Trees - "goss", uses Gradient-based One-Side Sampling - "rf", uses Random Forest
- **learning_rate** (*float*) – Boosting learning rate. Defaults to 0.1.
- **n_estimators** (*int*) – Number of boosted trees to fit. Defaults to 100.
- **max_depth** (*int*) – Maximum tree depth for base learners, <=0 means no limit. Defaults to 0.
- **num_leaves** (*int*) – Maximum tree leaves for base learners. Defaults to 31.
- **min_child_samples** (*int*) – Minimum number of data needed in a child (leaf). Defaults to 20.
- **bagging_fraction** (*float*) – LightGBM will randomly select a subset of features on each iteration (tree) without resampling if this is smaller than 1.0. For example, if set to 0.8, LightGBM will select 80% of features before training each tree. This can be used to speed up training and deal with overfitting. Defaults to 0.9.
- **bagging_freq** (*int*) – Frequency for bagging. 0 means bagging is disabled. k means perform bagging at every k iteration. Every k-th iteration, LightGBM will randomly select bagging_fraction * 100 % of the data to use for the next k iterations. Defaults to 0.
- **n_jobs** (*int or None*) – Number of threads to run in parallel. -1 uses all threads. Defaults to -1.
- **random_seed** (*int*) – Seed for the random number generator. Defaults to 0.

**Attributes**

| | |
|---|---|
| **hyper-parameter_ranges** | { "learning_rate": Real(0.000001, 1), "boosting_type": ["gbdt", "dart", "goss", "rf"], "n_estimators": Integer(10, 100), "max_depth": Integer(0, 10), "num_leaves": Integer(2, 100), "min_child_samples": Integer(1, 100), "bagging_fraction": Real(0.000001, 1), "bagging_freq": Integer(0, 1),} |
| **model_family** | ModelFamily.LIGHTGBM |
| **modifies_features** | True |
| **modifies_target** | False |
| **name** | LightGBM Regressor |
| **SEED_MAX** | SEED_BOUNDS.max_bound |
| **SEED_MIN** | 0 |
| **supported_problem_types** | [ProblemTypes.REGRESSION] |
| **training_only** | False |

**Methods**

| | |
|---|---|
| *clone* | Constructs a new component with the same parameters and random state. |
| *default_parameters* | Returns the default parameters for this component. |
| *describe* | Describe a component and its parameters. |
| *feature_importance* | Returns importance associated with each feature. |
| *fit* | Fits LightGBM regressor to data. |
| *get_prediction_intervals* | Find the prediction intervals using the fitted regressor. |
| *load* | Loads component at file path. |
| *needs_fitting* | Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances. |
| *parameters* | Returns the parameters which were used to initialize the component. |
| *predict* | Make predictions using fitted LightGBM regressor. |
| *predict_proba* | Make probability estimates for labels. |
| *save* | Saves component at file path. |
| *update_parameters* | Updates the parameter dictionary of the component. |

**clone**(*self*)

Constructs a new component with the same parameters and random state.

**Returns** A new instance of this component with identical parameters and random state.

**default_parameters**(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

**Returns** Default parameters for this component.

**Return type** dict

**describe**(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

**Parameters**

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description as dictionary in the format {"name": name, "parameters": parameters}

**Returns** Returns dictionary if return_dict is True, else None.

**Return type** None or dict

**property feature_importance**(*self*) → pandas.Series

Returns importance associated with each feature.

**Returns** Importance associated with each feature.

**Return type** np.ndarray

**Raises** **MethodPropertyNotFoundError** – If estimator does not have a feature_importance method or a component_obj that implements feature_importance.

**fit**(*self*, *X*, *y=None*)

Fits LightGBM regressor to data.

**Parameters**

- **X** (*pd.DataFrame*) – The input training data of shape [n_samples, n_features].
- **y** (*pd.Series*) – The target training data of length [n_samples].

**Returns** self

**get_prediction_intervals**(*self, X: pandas.DataFrame, y: Optional[pandas.Series] = None, coverage: List[float] = None, predictions: pandas.Series = None*) → Dict[str, pandas.Series]

Find the prediction intervals using the fitted regressor.

This function takes the predictions of the fitted estimator and calculates the rolling standard deviation across all predictions using a window size of 5. The lower and upper predictions are determined by taking the percent point (quantile) function of the lower tail probability at each bound multiplied by the rolling standard deviation.

**Parameters**

- **X** (*pd.DataFrame*) – Data of shape [n_samples, n_features].
- **y** (*pd.Series*) – Target data. Ignored.
- **coverage** (*list[float]*) – A list of floats between the values 0 and 1 that the upper and lower bounds of the prediction interval should be calculated for.
- **predictions** (*pd.Series*) – Optional list of predictions to use. If None, will generate predictions using *X*.

**Returns** Prediction intervals, keys are in the format {coverage}_lower or {coverage}_upper.

**Return type** dict

**Raises** **MethodPropertyNotFoundError** – If the estimator does not support Time Series Regression as a problem type.

**static load**(*file_path*)

Loads component at file path.

**Parameters** **file_path** (*str*) – Location to load file.

**Returns** ComponentBase object

**needs_fitting**(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need to be fit or whose fit methods do nothing.

**Returns** True.

**property parameters**(*self*)

Returns the parameters which were used to initialize the component.

**predict**(*self, X*)

Make predictions using fitted LightGBM regressor.

**Parameters** **X** (*pd.DataFrame*) – Data of shape [n_samples, n_features].

**Returns** Predicted values.

**Return type** pd.Series

**predict_proba**(*self*, *X: pandas.DataFrame*) → pandas.Series

Make probability estimates for labels.

**Parameters** **X** (*pd.DataFrame*) – Features.

**Returns** Probability estimates.

**Return type** pd.Series

**Raises** **MethodPropertyNotFoundError** – If estimator does not have a predict_proba method or a component_obj that implements predict_proba.

**save**(*self*, *file_path*, *pickle_protocol=cloudpickle.DEFAULT_PROTOCOL*)

Saves component at file path.

**Parameters**

- **file_path** (*str*) – Location to save file.
- **pickle_protocol** (*int*) – The pickle data stream format.

**update_parameters**(*self*, *update_dict*, *reset_fit=True*)

Updates the parameter dictionary of the component.

**Parameters**

- **update_dict** (*dict*) – A dict of parameters to update.
- **reset_fit** (*bool, optional*) – If True, will set *_is_fitted* to False.

**class** evalml.pipelines.components.estimators.regressors.**LinearRegressor**(*fit_intercept=True*, *n_jobs=- 1*, *random_seed=0*, ***kwargs*)

Linear Regressor.

**Parameters**

- **fit_intercept** (*boolean*) – Whether to calculate the intercept for this model. If set to False, no intercept will be used in calculations (i.e. data is expected to be centered). Defaults to True.
- **n_jobs** (*int or None*) – Number of jobs to run in parallel. -1 uses all threads. Defaults to -1.
- **random_seed** (*int*) – Seed for the random number generator. Defaults to 0.

**Attributes**

| | |
|---|---|
| **hyper-parameter_ranges** | { "fit_intercept": [True, False],} |
| **model_family** | ModelFamily.LINEAR_MODEL |
| **modifies_features** | True |
| **modifies_target** | False |
| **name** | Linear Regressor |
| **supported_problem_types** | [ ProblemTypes.REGRESSION, ProblemTypes.TIME_SERIES_REGRESSION,] |
| **training_only** | False |

**Methods**

| | |
|---|---|
| *clone* | Constructs a new component with the same parameters and random state. |
| *default_parameters* | Returns the default parameters for this component. |
| *describe* | Describe a component and its parameters. |
| *feature_importance* | Feature importance for fitted linear regressor. |
| *fit* | Fits estimator to data. |
| *get_prediction_intervals* | Find the prediction intervals using the fitted regressor. |
| *load* | Loads component at file path. |
| *needs_fitting* | Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances. |
| *parameters* | Returns the parameters which were used to initialize the component. |
| *predict* | Make predictions using selected features. |
| *predict_proba* | Make probability estimates for labels. |
| *save* | Saves component at file path. |
| *update_parameters* | Updates the parameter dictionary of the component. |

**clone**(*self*)

Constructs a new component with the same parameters and random state.

**Returns** A new instance of this component with identical parameters and random state.

**default_parameters**(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

**Returns** Default parameters for this component.

**Return type** dict

**describe**(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

**Parameters**

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description as dictionary in the format {"name": name, "parameters": parameters}

**Returns** Returns dictionary if return_dict is True, else None.

**Return type** None or dict

**property feature_importance**(*self*)

Feature importance for fitted linear regressor.

**fit**(*self*, *X: pandas.DataFrame*, *y: Optional[pandas.Series] = None*)

Fits estimator to data.

**Parameters**

- **X** (*pd.DataFrame*) – The input training data of shape [n_samples, n_features].
- **y** (*pd.Series, optional*) – The target training data of length [n_samples].

**Returns** self

**get_prediction_intervals**(*self*, *X: pandas.DataFrame*, *y: Optional[pandas.Series] = None*, *coverage: List[float] = None*, *predictions: pandas.Series = None*) → Dict[str, pandas.Series]

Find the prediction intervals using the fitted regressor.

This function takes the predictions of the fitted estimator and calculates the rolling standard deviation across all predictions using a window size of 5. The lower and upper predictions are determined by taking the percent point (quantile) function of the lower tail probability at each bound multiplied by the rolling standard deviation.

**Parameters**

- **X** (*pd.DataFrame*) – Data of shape [n_samples, n_features].
- **y** (*pd.Series*) – Target data. Ignored.
- **coverage** (*list[float]*) – A list of floats between the values 0 and 1 that the upper and lower bounds of the prediction interval should be calculated for.
- **predictions** (*pd.Series*) – Optional list of predictions to use. If None, will generate predictions using *X*.

**Returns** Prediction intervals, keys are in the format {coverage}_lower or {coverage}_upper.

**Return type** dict

**Raises** **MethodPropertyNotFoundError** – If the estimator does not support Time Series Regression as a problem type.

**static load**(*file_path*)

Loads component at file path.

**Parameters** **file_path** (*str*) – Location to load file.

**Returns** ComponentBase object

**needs_fitting**(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need to be fit or whose fit methods do nothing.

**Returns** True.

**property parameters**(*self*)

Returns the parameters which were used to initialize the component.

**predict**(*self*, *X: pandas.DataFrame*) → pandas.Series

Make predictions using selected features.

**Parameters** **X** (*pd.DataFrame*) – Data of shape [n_samples, n_features].

**Returns** Predicted values.

**Return type** pd.Series

**Raises** **MethodPropertyNotFoundError** – If estimator does not have a predict method or a component_obj that implements predict.

**predict_proba**(*self*, *X: pandas.DataFrame*) → pandas.Series

Make probability estimates for labels.

**Parameters** **X** (*pd.DataFrame*) – Features.

**Returns** Probability estimates.

**Return type** pd.Series

**Raises** **MethodPropertyNotFoundError** – If estimator does not have a predict_proba method or a component_obj that implements predict_proba.

**save**(*self*, *file_path*, *pickle_protocol=cloudpickle.DEFAULT_PROTOCOL*)

Saves component at file path.

**Parameters**

- **file_path** (*str*) – Location to save file.
- **pickle_protocol** (*int*) – The pickle data stream format.

**update_parameters**(*self*, *update_dict*, *reset_fit=True*)

Updates the parameter dictionary of the component.

**Parameters**

- **update_dict** (*dict*) – A dict of parameters to update.
- **reset_fit** (*bool, optional*) – If True, will set *_is_fitted* to False.

**class** evalml.pipelines.components.estimators.regressors.**ProphetRegressor**(*time_index: Optional[Hashable] = None*, *changepoint_prior_scale: float = 0.05*, *seasonality_prior_scale: int = 10*, *holidays_prior_scale: int = 10*, *seasonality_mode: str = 'additive'*, *stan_backend: str = 'CMDSTANPY'*, *interval_width: float = 0.95*, *random_seed: Union[int, float] = 0*, ***kwargs*)

Prophet is a procedure for forecasting time series data based on an additive model where non-linear trends are fit with yearly, weekly, and daily seasonality, plus holiday effects. It works best with time series that have strong seasonal effects and several seasons of historical data. Prophet is robust to missing data and shifts in the trend, and typically handles outliers well.

More information here: https://facebook.github.io/prophet/

**Parameters**

- **time_index** (*str*) – Specifies the name of the column in X that provides the datetime objects. Defaults to None.

- **changepoint_prior_scale** (*float*) – Determines the strength of the sparse prior for fitting on rate changes. Increasing this value increases the flexibility of the trend. Defaults to 0.05.
- **seasonality_prior_scale** (*int*) – Similar to changepoint_prior_scale. Adjusts the extent to which the seasonality model will fit the data. Defaults to 10.
- **holidays_prior_scale** (*int*) – Similar to changepoint_prior_scale. Adjusts the extent to which holidays will fit the data. Defaults to 10.
- **seasonality_mode** (*str*) – Determines how this component fits the seasonality. Options are "additive" and "multiplicative". Defaults to "additive".
- **stan_backend** (*str*) – Determines the backend that should be used to run Prophet. Options are "CMDSTANPY" and "PYSTAN". Defaults to "CMDSTANPY".
- **interval_width** (*float*) – Determines the confidence of the prediction interval range when calling *get_prediction_intervals*. Accepts values in the range (0,1). Defaults to 0.95.
- **random_seed** (*int*) – Seed for the random number generator. Defaults to 0.

**Attributes**

| | |
|---|---|
| **hyperparameter_ranges** | { "changepoint_prior_scale": Real(0.001, 0.5), "seasonality_prior_scale": Real(0.01, 10), "holidays_prior_scale": Real(0.01, 10), "seasonality_mode": ["additive", "multiplicative"],} |
| **model_family** | ModelFamily.PROPHET |
| **modifies_features** | True |
| **modifies_target** | False |
| **name** | Prophet Regressor |
| **supported_problem_types** | [ProblemTypes.TIME_SERIES_REGRESSION] |
| **training_only** | False |

**Methods**

| | |
|---|---|
| *build_prophet_df* | Build the Prophet data to pass fit and predict on. |
| *clone* | Constructs a new component with the same parameters and random state. |
| *default_parameters* | Returns the default parameters for this component. |
| *describe* | Describe a component and its parameters. |
| *feature_importance* | Returns array of 0's with len(1) as feature_importance is not defined for Prophet regressor. |
| *fit* | Fits Prophet regressor component to data. |
| *get_params* | Get parameters for the Prophet regressor. |
| *get_prediction_intervals* | Find the prediction intervals using the fitted ProphetRegressor. |
| *load* | Loads component at file path. |
| *needs_fitting* | Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances. |
| *parameters* | Returns the parameters which were used to initialize the component. |
| *predict* | Make predictions using fitted Prophet regressor. |
| *predict_proba* | Make probability estimates for labels. |
| *save* | Saves component at file path. |
| *update_parameters* | Updates the parameter dictionary of the component. |

**static build_prophet_df**(*X: pandas.DataFrame*, *y: Optional[pandas.Series] = None*, *time_index: str = 'ds'*) → pandas.DataFrame

Build the Prophet data to pass fit and predict on.

**clone**(*self*)

Constructs a new component with the same parameters and random state.

**Returns** A new instance of this component with identical parameters and random state.

**default_parameters**(*cls*) → dict

Returns the default parameters for this component.

**Returns** Default parameters for this component.

**Return type** dict

**describe**(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

**Parameters**

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description as dictionary in the format {"name": name, "parameters": parameters}

**Returns** Returns dictionary if return_dict is True, else None.

**Return type** None or dict

**property feature_importance**(*self*) → numpy.ndarray

Returns array of 0's with len(1) as feature_importance is not defined for Prophet regressor.

**fit**(*self*, *X: pandas.DataFrame*, *y: Optional[pandas.Series] = None*)

Fits Prophet regressor component to data.

**Parameters**

- **X** (*pd.DataFrame*) – The input training data of shape [n_samples, n_features].
- **y** (*pd.Series*) – The target training data of length [n_samples].

**Returns** self

**get_params**(*self*) → dict

Get parameters for the Prophet regressor.

**get_prediction_intervals**(*self*, *X: pandas.DataFrame*, *y: Optional[pandas.Series] = None*, *coverage: List[float] = None*, *predictions: pandas.Series = None*) → Dict[str, pandas.Series]

Find the prediction intervals using the fitted ProphetRegressor.

**Parameters**

- **X** (*pd.DataFrame*) – Data of shape [n_samples, n_features].
- **y** (*pd.Series*) – Target data. Ignored.
- **coverage** (*List[float]*) – A list of floats between the values 0 and 1 that the upper and lower bounds of the prediction interval should be calculated for.
- **predictions** (*pd.Series*) – Not used for Prophet estimator.

**Returns** Prediction intervals, keys are in the format {coverage}_lower or {coverage}_upper.

**Return type** dict

**static load**(*file_path*)

Loads component at file path.

**Parameters** **file_path** (*str*) – Location to load file.

**Returns** ComponentBase object

**needs_fitting**(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need to be fit or whose fit methods do nothing.

**Returns** True.

**property parameters**(*self*)

Returns the parameters which were used to initialize the component.

**predict**(*self*, *X: pandas.DataFrame*, *y: Optional[pandas.Series] = None*) → pandas.Series

Make predictions using fitted Prophet regressor.

**Parameters**

- **X** (*pd.DataFrame*) – Data of shape [n_samples, n_features].
- **y** (*pd.Series*) – Target data. Ignored.

**Returns** Predicted values.

**Return type** pd.Series

**predict_proba**(*self*, *X: pandas.DataFrame*) → pandas.Series

Make probability estimates for labels.

**Parameters** **X** (*pd.DataFrame*) – Features.

**Returns** Probability estimates.

**Return type** pd.Series

**Raises** **MethodPropertyNotFoundError** – If estimator does not have a predict_proba method or a component_obj that implements predict_proba.

**save**(*self*, *file_path*, *pickle_protocol=cloudpickle.DEFAULT_PROTOCOL*)

Saves component at file path.

**Parameters**

- **file_path** (*str*) – Location to save file.
- **pickle_protocol** (*int*) – The pickle data stream format.

**update_parameters**(*self*, *update_dict*, *reset_fit=True*)

Updates the parameter dictionary of the component.

**Parameters**

- **update_dict** (*dict*) – A dict of parameters to update.
- **reset_fit** (*bool, optional*) – If True, will set *_is_fitted* to False.

**class** evalml.pipelines.components.estimators.regressors.**RandomForestRegressor**(*n_estimators: int = 100, max_depth: int = 6, n_jobs: int = -1, random_seed: Union[int, float] = 0, **kwargs*)

Random Forest Regressor.

**Parameters**

- **n_estimators** (*float*) – The number of trees in the forest. Defaults to 100.
- **max_depth** (*int*) – Maximum tree depth for base learners. Defaults to 6.
- **n_jobs** (*int or None*) – Number of jobs to run in parallel. -1 uses all processes. Defaults to -1.
- **random_seed** (*int*) – Seed for the random number generator. Defaults to 0.

**Attributes**

| | |
|---|---|
| **hyper-parameter_ranges** | { "n_estimators": Integer(10, 1000), "max_depth": Integer(1, 32),} |
| **model_family** | ModelFamily.RANDOM_FOREST |
| **modifies_features** | True |
| **modifies_target** | False |
| **name** | Random Forest Regressor |
| **supported_problem_types** | [ ProblemTypes.REGRESSION, ProblemTypes.TIME_SERIES_REGRESSION,] |
| **training_only** | False |

**Methods**

| | |
|---|---|
| *clone* | Constructs a new component with the same parameters and random state. |
| *default_parameters* | Returns the default parameters for this component. |
| *describe* | Describe a component and its parameters. |
| *feature_importance* | Returns importance associated with each feature. |
| *fit* | Fits estimator to data. |
| *get_prediction_intervals* | Find the prediction intervals using the fitted RandomForestRegressor. |
| *load* | Loads component at file path. |
| *needs_fitting* | Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances. |
| *parameters* | Returns the parameters which were used to initialize the component. |
| *predict* | Make predictions using selected features. |
| *predict_proba* | Make probability estimates for labels. |
| *save* | Saves component at file path. |
| *update_parameters* | Updates the parameter dictionary of the component. |

**clone**(*self*)

Constructs a new component with the same parameters and random state.

**Returns** A new instance of this component with identical parameters and random state.

**default_parameters**(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

**Returns** Default parameters for this component.

**Return type** dict

**describe**(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

**Parameters**

- **print_name** (*bool, optional*) – whether to print name of component

- **return_dict** (*bool, optional*) – whether to return description as dictionary in the format {"name": name, "parameters": parameters}

**Returns** Returns dictionary if return_dict is True, else None.

**Return type** None or dict

**property feature_importance**(*self*) → pandas.Series

Returns importance associated with each feature.

**Returns** Importance associated with each feature.

**Return type** np.ndarray

**Raises MethodPropertyNotFoundError** – If estimator does not have a feature_importance method or a component_obj that implements feature_importance.

**fit**(*self, X: pandas.DataFrame, y: Optional[pandas.Series] = None*)

Fits estimator to data.

**Parameters**

- **X** (*pd.DataFrame*) – The input training data of shape [n_samples, n_features].
- **y** (*pd.Series, optional*) – The target training data of length [n_samples].

**Returns** self

**get_prediction_intervals**(*self, X: pandas.DataFrame, y: Optional[pandas.Series] = None, coverage: List[float] = None, predictions: pandas.Series = None*) → Dict[str, pandas.Series]

Find the prediction intervals using the fitted RandomForestRegressor.

**Parameters**

- **X** (*pd.DataFrame*) – Data of shape [n_samples, n_features].
- **y** (*pd.Series*) – Target data. Optional.
- **coverage** (*list[float]*) – A list of floats between the values 0 and 1 that the upper and lower bounds of the prediction interval should be calculated for.
- **predictions** (*pd.Series*) – Optional list of predictions to use. If None, will generate predictions using *X*.

**Returns** Prediction intervals, keys are in the format {coverage}_lower or {coverage}_upper.

**Return type** dict

**static load**(*file_path*)

Loads component at file path.

**Parameters file_path** (*str*) – Location to load file.

**Returns** ComponentBase object

**needs_fitting**(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need to be fit or whose fit methods do nothing.

**Returns** True.

**property parameters**(*self*)

Returns the parameters which were used to initialize the component.

**predict**(*self*, *X: pandas.DataFrame*) → pandas.Series

Make predictions using selected features.

**Parameters** **X** (*pd.DataFrame*) – Data of shape [n_samples, n_features].

**Returns** Predicted values.

**Return type** pd.Series

**Raises** **MethodPropertyNotFoundError** – If estimator does not have a predict method or a component_obj that implements predict.

**predict_proba**(*self*, *X: pandas.DataFrame*) → pandas.Series

Make probability estimates for labels.

**Parameters** **X** (*pd.DataFrame*) – Features.

**Returns** Probability estimates.

**Return type** pd.Series

**Raises** **MethodPropertyNotFoundError** – If estimator does not have a predict_proba method or a component_obj that implements predict_proba.

**save**(*self*, *file_path*, *pickle_protocol=cloudpickle.DEFAULT_PROTOCOL*)

Saves component at file path.

**Parameters**

- **file_path** (*str*) – Location to save file.
- **pickle_protocol** (*int*) – The pickle data stream format.

**update_parameters**(*self*, *update_dict*, *reset_fit=True*)

Updates the parameter dictionary of the component.

**Parameters**

- **update_dict** (*dict*) – A dict of parameters to update.
- **reset_fit** (*bool, optional*) – If True, will set *_is_fitted* to False.

**class** evalml.pipelines.components.estimators.regressors.**SVMRegressor**(*C=1.0*, *kernel='rbf'*, *gamma='auto'*, *random_seed=0*, ***kwargs*)

Support Vector Machine Regressor.

**Parameters**

- **C** (*float*) – The regularization parameter. The strength of the regularization is inversely proportional to C. Must be strictly positive. The penalty is a squared l2 penalty. Defaults to 1.0.
- **kernel** (*{"poly", "rbf", "sigmoid"}*) – Specifies the kernel type to be used in the algorithm. Defaults to "rbf".
- **gamma** (*{"scale", "auto"} or float*) – Kernel coefficient for "rbf", "poly" and "sigmoid". Defaults to "auto". - If gamma='scale' is passed then it uses 1 / (n_features * X.var()) as value of gamma - If "auto" (default), uses 1 / n_features

- **random_seed** (*int*) – Seed for the random number generator. Defaults to 0.

**Attributes**

| | |
|---|---|
| **hyper-parameter_ranges** | { "C": Real(0, 10), "kernel": ["poly", "rbf", "sigmoid"], "gamma": ["scale", "auto"],} |
| **model_family** | ModelFamily.SVM |
| **modifies_features** | True |
| **modifies_target** | False |
| **name** | SVM Regressor |
| **supported_problem_types** | [ ProblemTypes.REGRESSION, ProblemTypes.TIME_SERIES_REGRESSION,] |
| **training_only** | False |

**Methods**

| | |
|---|---|
| *clone* | Constructs a new component with the same parameters and random state. |
| *default_parameters* | Returns the default parameters for this component. |
| *describe* | Describe a component and its parameters. |
| *feature_importance* | Feature importance of fitted SVM regresor. |
| *fit* | Fits estimator to data. |
| *get_prediction_intervals* | Find the prediction intervals using the fitted regressor. |
| *load* | Loads component at file path. |
| *needs_fitting* | Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances. |
| *parameters* | Returns the parameters which were used to initialize the component. |
| *predict* | Make predictions using selected features. |
| *predict_proba* | Make probability estimates for labels. |
| *save* | Saves component at file path. |
| *update_parameters* | Updates the parameter dictionary of the component. |

**clone**(*self*)

Constructs a new component with the same parameters and random state.

**Returns** A new instance of this component with identical parameters and random state.

**default_parameters**(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

**Returns** Default parameters for this component.

**Return type** dict

**describe**(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

**Parameters**

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description as dictionary in the format {"name": name, "parameters": parameters}

**Returns** Returns dictionary if return_dict is True, else None.

**Return type** None or dict

**property feature_importance**(*self*)

Feature importance of fitted SVM regresor.

Only works with linear kernels. If the kernel isn't linear, we return a numpy array of zeros.

**Returns** The feature importance of the fitted SVM regressor, or an array of zeroes if the kernel is not linear.

**fit**(*self, X: pandas.DataFrame, y: Optional[pandas.Series] = None*)

Fits estimator to data.

**Parameters**

- **X** (*pd.DataFrame*) – The input training data of shape [n_samples, n_features].
- **y** (*pd.Series, optional*) – The target training data of length [n_samples].

**Returns** self

**get_prediction_intervals**(*self, X: pandas.DataFrame, y: Optional[pandas.Series] = None, coverage: List[float] = None, predictions: pandas.Series = None*) → Dict[str, pandas.Series]

Find the prediction intervals using the fitted regressor.

This function takes the predictions of the fitted estimator and calculates the rolling standard deviation across all predictions using a window size of 5. The lower and upper predictions are determined by taking the percent point (quantile) function of the lower tail probability at each bound multiplied by the rolling standard deviation.

**Parameters**

- **X** (*pd.DataFrame*) – Data of shape [n_samples, n_features].
- **y** (*pd.Series*) – Target data. Ignored.
- **coverage** (*list[float]*) – A list of floats between the values 0 and 1 that the upper and lower bounds of the prediction interval should be calculated for.
- **predictions** (*pd.Series*) – Optional list of predictions to use. If None, will generate predictions using *X*.

**Returns** Prediction intervals, keys are in the format {coverage}_lower or {coverage}_upper.

**Return type** dict

**Raises** **MethodPropertyNotFoundError** – If the estimator does not support Time Series Regression as a problem type.

**static load**(*file_path*)

Loads component at file path.

**Parameters** **file_path** (*str*) – Location to load file.

**Returns** ComponentBase object

**needs_fitting**(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need to be fit or whose fit methods do nothing.

**Returns** True.

**property parameters**(*self*)

Returns the parameters which were used to initialize the component.

**predict**(*self*, *X: pandas.DataFrame*) → pandas.Series

Make predictions using selected features.

**Parameters** **X** (*pd.DataFrame*) – Data of shape [n_samples, n_features].

**Returns** Predicted values.

**Return type** pd.Series

**Raises** **MethodPropertyNotFoundError** – If estimator does not have a predict method or a component_obj that implements predict.

**predict_proba**(*self*, *X: pandas.DataFrame*) → pandas.Series

Make probability estimates for labels.

**Parameters** **X** (*pd.DataFrame*) – Features.

**Returns** Probability estimates.

**Return type** pd.Series

**Raises** **MethodPropertyNotFoundError** – If estimator does not have a predict_proba method or a component_obj that implements predict_proba.

**save**(*self*, *file_path*, *pickle_protocol=cloudpickle.DEFAULT_PROTOCOL*)

Saves component at file path.

**Parameters**

- **file_path** (*str*) – Location to save file.
- **pickle_protocol** (*int*) – The pickle data stream format.

**update_parameters**(*self*, *update_dict*, *reset_fit=True*)

Updates the parameter dictionary of the component.

**Parameters**

- **update_dict** (*dict*) – A dict of parameters to update.
- **reset_fit** (*bool, optional*) – If True, will set *_is_fitted* to False.

**class** evalml.pipelines.components.estimators.regressors.**TimeSeriesBaselineEstimator**(*gap=1*, *forecast_horizon=1*, *random_seed=0*, ***kwargs*)

Time series estimator that predicts using the naive forecasting approach.

This is useful as a simple baseline estimator for time series problems.

**Parameters**

- **gap** (*int*) – Gap between prediction date and target date and must be a positive integer. If gap is 0, target date will be shifted ahead by 1 time period. Defaults to 1.
- **forecast_horizon** (*int*) – Number of time steps the model is expected to predict.
- **random_seed** (*int*) – Seed for the random number generator. Defaults to 0.

**Attributes**

| | |
|---|---|
| **hyper-parameter_ranges** | {} |
| **model_family** | ModelFamily.BASELINE |
| **modifies_features** | True |
| **modifies_target** | False |
| **name** | Time Series Baseline Estimator |
| **supported_problem_types** | [ ProblemTypes.TIME_SERIES_REGRESSION, ProblemTypes.TIME_SERIES_BINARY, ProblemTypes.TIME_SERIES_MULTICLASS,] |
| **training_only** | False |

**Methods**

| | |
|---|---|
| *clone* | Constructs a new component with the same parameters and random state. |
| *default_parameters* | Returns the default parameters for this component. |
| *describe* | Describe a component and its parameters. |
| *feature_importance* | Returns importance associated with each feature. |
| *fit* | Fits time series baseline estimator to data. |
| *get_prediction_intervals* | Find the prediction intervals using the fitted regressor. |
| *load* | Loads component at file path. |
| *needs_fitting* | Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances. |
| *parameters* | Returns the parameters which were used to initialize the component. |
| *predict* | Make predictions using fitted time series baseline estimator. |
| *predict_proba* | Make prediction probabilities using fitted time series baseline estimator. |
| *save* | Saves component at file path. |
| *update_parameters* | Updates the parameter dictionary of the component. |

**clone**(*self*)

Constructs a new component with the same parameters and random state.

**Returns** A new instance of this component with identical parameters and random state.

**default_parameters**(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

**Returns** Default parameters for this component.

**Return type** dict

`describe`(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

**Parameters**

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description as dictionary in the format {"name": name, "parameters": parameters}

**Returns** Returns dictionary if return_dict is True, else None.

**Return type** None or dict

`property feature_importance`(*self*)

Returns importance associated with each feature.

Since baseline estimators do not use input features to calculate predictions, returns an array of zeroes.

**Returns** An array of zeroes.

**Return type** np.ndarray (float)

`fit`(*self*, *X*, *y=None*)

Fits time series baseline estimator to data.

**Parameters**

- **X** (*pd.DataFrame*) – The input training data of shape [n_samples, n_features].
- **y** (*pd.Series*) – The target training data of length [n_samples].

**Returns** self

**Raises** **ValueError** – If input y is None.

`get_prediction_intervals`(*self*, *X: pandas.DataFrame*, *y: Optional[pandas.Series] = None*, *coverage: List[float] = None*, *predictions: pandas.Series = None*) → Dict[str, pandas.Series]

Find the prediction intervals using the fitted regressor.

This function takes the predictions of the fitted estimator and calculates the rolling standard deviation across all predictions using a window size of 5. The lower and upper predictions are determined by taking the percent point (quantile) function of the lower tail probability at each bound multiplied by the rolling standard deviation.

**Parameters**

- **X** (*pd.DataFrame*) – Data of shape [n_samples, n_features].
- **y** (*pd.Series*) – Target data. Ignored.
- **coverage** (*list[float]*) – A list of floats between the values 0 and 1 that the upper and lower bounds of the prediction interval should be calculated for.
- **predictions** (*pd.Series*) – Optional list of predictions to use. If None, will generate predictions using *X*.

**Returns** Prediction intervals, keys are in the format {coverage}_lower or {coverage}_upper.

**Return type** dict

**Raises** `MethodPropertyNotFoundError` – If the estimator does not support Time Series Regression as a problem type.

`static load`(*file_path*)

Loads component at file path.

**Parameters** `file_path` (*str*) – Location to load file.

**Returns** ComponentBase object

`needs_fitting`(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need to be fit or whose fit methods do nothing.

**Returns** True.

`property parameters`(*self*)

Returns the parameters which were used to initialize the component.

`predict`(*self*, *X*)

Make predictions using fitted time series baseline estimator.

**Parameters** `X` (*pd.DataFrame*) – Data of shape [n_samples, n_features].

**Returns** Predicted values.

**Return type** pd.Series

**Raises** `ValueError` – If input y is None.

`predict_proba`(*self*, *X*)

Make prediction probabilities using fitted time series baseline estimator.

**Parameters** `X` (*pd.DataFrame*) – Data of shape [n_samples, n_features].

**Returns** Predicted probability values.

**Return type** pd.DataFrame

**Raises** `ValueError` – If input y is None.

`save`(*self*, *file_path*, *pickle_protocol=cloudpickle.DEFAULT_PROTOCOL*)

Saves component at file path.

**Parameters**

- `file_path` (*str*) – Location to save file.
- `pickle_protocol` (*int*) – The pickle data stream format.

`update_parameters`(*self*, *update_dict*, *reset_fit=True*)

Updates the parameter dictionary of the component.

**Parameters**

- `update_dict` (*dict*) – A dict of parameters to update.
- `reset_fit` (*bool, optional*) – If True, will set *_is_fitted* to False.

**class** evalml.pipelines.components.estimators.regressors.**VowpalWabbitRegressor**(*learning_rate=0.5, decay_learning_rate=1.0, power_t=0.5, passes=1, random_seed=0, **kwargs*)

Vowpal Wabbit Regressor.

**Parameters**

- **learning_rate** (*float*) – Boosting learning rate. Defaults to 0.5.
- **decay_learning_rate** (*float*) – Decay factor for learning_rate. Defaults to 1.0.
- **power_t** (*float*) – Power on learning rate decay. Defaults to 0.5.
- **passes** (*int*) – Number of training passes. Defaults to 1.
- **random_seed** (*int*) – Seed for the random number generator. Defaults to 0.

**Attributes**

| | |
|---|---|
| **hyper-parameter_ranges** | None |
| **model_family** | ModelFamily.VOWPAL_WABBIT |
| **modifies_features** | True |
| **modifies_target** | False |
| **name** | Vowpal Wabbit Regressor |
| **supported_problem_types** | [ ProblemTypes.REGRESSION, ProblemTypes.TIME_SERIES_REGRESSION,] |
| **training_only** | False |

**Methods**

| | |
|---|---|
| *clone* | Constructs a new component with the same parameters and random state. |
| *default_parameters* | Returns the default parameters for this component. |
| *describe* | Describe a component and its parameters. |
| *feature_importance* | Feature importance for Vowpal Wabbit regressor. |
| *fit* | Fits estimator to data. |
| *get_prediction_intervals* | Find the prediction intervals using the fitted regressor. |
| *load* | Loads component at file path. |
| *needs_fitting* | Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances. |
| *parameters* | Returns the parameters which were used to initialize the component. |
| *predict* | Make predictions using selected features. |
| *predict_proba* | Make probability estimates for labels. |
| *save* | Saves component at file path. |
| *update_parameters* | Updates the parameter dictionary of the component. |

**clone**(*self*)

Constructs a new component with the same parameters and random state.

**Returns** A new instance of this component with identical parameters and random state.

**default_parameters**(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

**Returns** Default parameters for this component.

**Return type** dict

**describe**(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

**Parameters**

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description as dictionary in the format {"name": name, "parameters": parameters}

**Returns** Returns dictionary if return_dict is True, else None.

**Return type** None or dict

**property feature_importance**(*self*)

Feature importance for Vowpal Wabbit regressor.

**fit**(*self*, *X: pandas.DataFrame*, *y: Optional[pandas.Series] = None*)

Fits estimator to data.

**Parameters**

- **X** (*pd.DataFrame*) – The input training data of shape [n_samples, n_features].
- **y** (*pd.Series, optional*) – The target training data of length [n_samples].

**Returns** self

**get_prediction_intervals**(*self*, *X: pandas.DataFrame*, *y: Optional[pandas.Series] = None*, *coverage: List[float] = None*, *predictions: pandas.Series = None*) → Dict[str, pandas.Series]

Find the prediction intervals using the fitted regressor.

This function takes the predictions of the fitted estimator and calculates the rolling standard deviation across all predictions using a window size of 5. The lower and upper predictions are determined by taking the percent point (quantile) function of the lower tail probability at each bound multiplied by the rolling standard deviation.

**Parameters**

- **X** (*pd.DataFrame*) – Data of shape [n_samples, n_features].
- **y** (*pd.Series*) – Target data. Ignored.
- **coverage** (*list[float]*) – A list of floats between the values 0 and 1 that the upper and lower bounds of the prediction interval should be calculated for.
- **predictions** (*pd.Series*) – Optional list of predictions to use. If None, will generate predictions using *X*.

**Returns** Prediction intervals, keys are in the format {coverage}_lower or {coverage}_upper.

**Return type** dict

**Raises** **MethodPropertyNotFoundError** – If the estimator does not support Time Series Regression as a problem type.

**static load**(*file_path*)

Loads component at file path.

**Parameters** **file_path** (*str*) – Location to load file.

**Returns** ComponentBase object

**needs_fitting**(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need to be fit or whose fit methods do nothing.

**Returns** True.

**property parameters**(*self*)

Returns the parameters which were used to initialize the component.

**predict**(*self*, *X: pandas.DataFrame*) → pandas.Series

Make predictions using selected features.

**Parameters** **X** (*pd.DataFrame*) – Data of shape [n_samples, n_features].

**Returns** Predicted values.

**Return type** pd.Series

**Raises** **MethodPropertyNotFoundError** – If estimator does not have a predict method or a component_obj that implements predict.

**predict_proba**(*self*, *X: pandas.DataFrame*) → pandas.Series

Make probability estimates for labels.

**Parameters** **X** (*pd.DataFrame*) – Features.

**Returns** Probability estimates.

**Return type** pd.Series

**Raises** `MethodPropertyNotFoundError` – If estimator does not have a predict_proba method or a component_obj that implements predict_proba.

`save`(*self*, *file_path*, *pickle_protocol=cloudpickle.DEFAULT_PROTOCOL*)

Saves component at file path.

**Parameters**

- `file_path` (*str*) – Location to save file.
- `pickle_protocol` (*int*) – The pickle data stream format.

`update_parameters`(*self*, *update_dict*, *reset_fit=True*)

Updates the parameter dictionary of the component.

**Parameters**

- `update_dict` (*dict*) – A dict of parameters to update.
- `reset_fit` (*bool, optional*) – If True, will set *_is_fitted* to False.

`class evalml.pipelines.components.estimators.regressors.XGBoostRegressor`(*eta: float = 0.1*, *max_depth: int = 6*, *min_child_weight: int = 1*, *n_estimators: int = 100*, *random_seed: Union[int, float] = 0*, *n_jobs: int = 12*, ***kwargs*)

XGBoost Regressor.

**Parameters**

- `eta` (*float*) – Boosting learning rate. Defaults to 0.1.
- `max_depth` (*int*) – Maximum tree depth for base learners. Defaults to 6.
- `min_child_weight` (*float*) – Minimum sum of instance weight (hessian) needed in a child. Defaults to 1.0
- `n_estimators` (*int*) – Number of gradient boosted trees. Equivalent to number of boosting rounds. Defaults to 100.
- `random_seed` (*int*) – Seed for the random number generator. Defaults to 0.
- `n_jobs` (*int*) – Number of parallel threads used to run xgboost. Note that creating thread contention will significantly slow down the algorithm. Defaults to 12.

**Attributes**

| | |
|---|---|
| **hyper-parameter_ranges** | { "eta": Real(0.000001, 1), "max_depth": Integer(1, 20), "min_child_weight": Real(1, 10), "n_estimators": Integer(1, 1000),} |
| **model_family** | ModelFamily.XGBOOST |
| **modifies_features** | True |
| **modifies_target** | False |
| **name** | XGBoost Regressor |
| **SEED_MAX** | None |
| **SEED_MIN** | None |
| **supported_problem_types** | [ ProblemTypes.REGRESSION, ProblemTypes.TIME_SERIES_REGRESSION,] |
| **training_only** | False |

**Methods**

| | |
|---|---|
| *clone* | Constructs a new component with the same parameters and random state. |
| *default_parameters* | Returns the default parameters for this component. |
| *describe* | Describe a component and its parameters. |
| *feature_importance* | Feature importance of fitted XGBoost regressor. |
| *fit* | Fits XGBoost regressor component to data. |
| *get_prediction_intervals* | Find the prediction intervals using the fitted XGBoostRegressor. |
| *load* | Loads component at file path. |
| *needs_fitting* | Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances. |
| *parameters* | Returns the parameters which were used to initialize the component. |
| *predict* | Make predictions using fitted XGBoost regressor. |
| *predict_proba* | Make probability estimates for labels. |
| *save* | Saves component at file path. |
| *update_parameters* | Updates the parameter dictionary of the component. |

**clone**(*self*)

Constructs a new component with the same parameters and random state.

**Returns** A new instance of this component with identical parameters and random state.

**default_parameters**(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

**Returns** Default parameters for this component.

**Return type** dict

**describe**(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

**Parameters**

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description as dictionary in the format {"name": name, "parameters": parameters}

**Returns** Returns dictionary if return_dict is True, else None.

**Return type** None or dict

**property feature_importance**(*self*) → pandas.Series

Feature importance of fitted XGBoost regressor.

**fit**(*self*, *X: pandas.DataFrame*, *y: Optional[pandas.Series] = None*)

Fits XGBoost regressor component to data.

**Parameters**

- **X** (*pd.DataFrame*) – The input training data of shape [n_samples, n_features].
- **y** (*pd.Series, optional*) – The target training data of length [n_samples].

**Returns** self

**get_prediction_intervals**(*self*, *X: pandas.DataFrame*, *y: Optional[pandas.Series] = None*, *coverage: List[float] = None*, *predictions: pandas.Series = None*) → Dict[str, pandas.Series]

Find the prediction intervals using the fitted XGBoostRegressor.

**Parameters**

- **X** (*pd.DataFrame*) – Data of shape [n_samples, n_features].
- **y** (*pd.Series*) – Target data. Ignored.
- **coverage** (*List[float]*) – A list of floats between the values 0 and 1 that the upper and lower bounds of the prediction interval should be calculated for.
- **predictions** (*pd.Series*) – Optional list of predictions to use. If None, will generate predictions using *X*.

**Returns** Prediction intervals, keys are in the format {coverage}_lower or {coverage}_upper.

**Return type** dict

**static load**(*file_path*)

Loads component at file path.

**Parameters** **file_path** (*str*) – Location to load file.

**Returns** ComponentBase object

**needs_fitting**(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need to be fit or whose fit methods do nothing.

**Returns** True.

**property parameters**(*self*)

Returns the parameters which were used to initialize the component.

**predict**(*self*, *X: pandas.DataFrame*) → pandas.Series

Make predictions using fitted XGBoost regressor.

**Parameters** **X** (*pd.DataFrame*) – Data of shape [n_samples, n_features].

**Returns** Predicted values.

**Return type** pd.Series

**predict_proba**(*self*, *X: pandas.DataFrame*) → pandas.Series

Make probability estimates for labels.

**Parameters** **X** (*pd.DataFrame*) – Features.

**Returns** Probability estimates.

**Return type** pd.Series

**Raises** **MethodPropertyNotFoundError** – If estimator does not have a predict_proba method or a component_obj that implements predict_proba.

**save**(*self*, *file_path*, *pickle_protocol=cloudpickle.DEFAULT_PROTOCOL*)

Saves component at file path.

**Parameters**

- **file_path** (*str*) – Location to save file.
- **pickle_protocol** (*int*) – The pickle data stream format.

**update_parameters**(*self*, *update_dict*, *reset_fit=True*)

Updates the parameter dictionary of the component.

**Parameters**

- **update_dict** (*dict*) – A dict of parameters to update.
- **reset_fit** (*bool, optional*) – If True, will set *_is_fitted* to False.

## Submodules

## estimator

A component that fits and predicts given data.

## Module Contents

## Classes Summary

| | |
|---|---|
| *Estimator* | A component that fits and predicts given data. |

## Contents

**class** evalml.pipelines.components.estimators.estimator.**Estimator**(*parameters: dict = None, component_obj: Type[evalml.pipelines.components.ComponentBas = None, random_seed: Union[int, float] = 0, **kwargs*)

A component that fits and predicts given data.

To implement a new Estimator, define your own class which is a subclass of Estimator, including a name and a list of acceptable ranges for any parameters to be tuned during the automl search (hyperparameters). Define an *__init__* method which sets up any necessary state and objects. Make sure your *__init__* only uses standard keyword arguments and calls *super().__init__()* with a parameters dict. You may also override the *fit*, *transform*, *fit_transform* and other methods in this class if appropriate.

To see some examples, check out the definitions of any Estimator component subclass.

**Parameters**

- **parameters** (*dict*) – Dictionary of parameters for the component. Defaults to None.
- **component_obj** (*obj*) – Third-party objects useful in component implementation. Defaults to None.
- **random_seed** (*int*) – Seed for the random number generator. Defaults to 0.

**Attributes**

| | |
|---|---|
| **model_family** | ModelFamily.NONE |
| **modifies_features** | True |
| **modifies_target** | False |
| **training_only** | False |

**Methods**

| | |
|---|---|
| clone | Constructs a new component with the same parameters and random state. |
| default_parameters | Returns the default parameters for this component. |
| describe | Describe a component and its parameters. |
| feature_importance | Returns importance associated with each feature. |
| fit | Fits estimator to data. |
| get_prediction_intervals | Find the prediction intervals using the fitted regressor. |
| load | Loads component at file path. |
| model_family | ModelFamily.NONE |
| name | Returns string name of this component. |
| needs_fitting | Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances. |
| parameters | Returns the parameters which were used to initialize the component. |
| predict | Make predictions using selected features. |
| predict_proba | Make probability estimates for labels. |
| save | Saves component at file path. |
| supported_problem_types | Problem types this estimator supports. |
| update_parameters | Updates the parameter dictionary of the component. |

**clone**(*self*)

Constructs a new component with the same parameters and random state.

**Returns** A new instance of this component with identical parameters and random state.

**default_parameters**(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

**Returns** Default parameters for this component.

**Return type** dict

**describe**(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

**Parameters**

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description as dictionary in the format {"name": name, "parameters": parameters}

**Returns** Returns dictionary if return_dict is True, else None.

**Return type** None or dict

**property feature_importance**(*self*) → pandas.Series

Returns importance associated with each feature.

**Returns** Importance associated with each feature.

**Return type** np.ndarray

**Raises** **MethodPropertyNotFoundError** – If estimator does not have a feature_importance method or a component_obj that implements feature_importance.

**fit**(*self*, *X: pandas.DataFrame*, *y: Optional[pandas.Series] = None*)

Fits estimator to data.

**Parameters**

- **X** (*pd.DataFrame*) – The input training data of shape [n_samples, n_features].
- **y** (*pd.Series, optional*) – The target training data of length [n_samples].

**Returns** self

**get_prediction_intervals**(*self*, *X: pandas.DataFrame*, *y: Optional[pandas.Series] = None*, *coverage: List[float] = None*, *predictions: pandas.Series = None*) → Dict[str, pandas.Series]

Find the prediction intervals using the fitted regressor.

This function takes the predictions of the fitted estimator and calculates the rolling standard deviation across all predictions using a window size of 5. The lower and upper predictions are determined by taking the percent point (quantile) function of the lower tail probability at each bound multiplied by the rolling standard deviation.

**Parameters**

- **X** (*pd.DataFrame*) – Data of shape [n_samples, n_features].
- **y** (*pd.Series*) – Target data. Ignored.
- **coverage** (*list[float]*) – A list of floats between the values 0 and 1 that the upper and lower bounds of the prediction interval should be calculated for.
- **predictions** (*pd.Series*) – Optional list of predictions to use. If None, will generate predictions using *X*.

**Returns** Prediction intervals, keys are in the format {coverage}_lower or {coverage}_upper.

**Return type** dict

**Raises** **MethodPropertyNotFoundError** – If the estimator does not support Time Series Regression as a problem type.

**static load**(*file_path*)

Loads component at file path.

**Parameters** **file_path** (*str*) – Location to load file.

**Returns** ComponentBase object

**property model_family**(*cls*)

Returns ModelFamily of this component.

**property name**(*cls*)

Returns string name of this component.

**needs_fitting**(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need to be fit or whose fit methods do nothing.

**Returns** True.

**property parameters**(*self*)

Returns the parameters which were used to initialize the component.

**predict**(*self*, *X: pandas.DataFrame*) → pandas.Series

Make predictions using selected features.

**Parameters** **X** (*pd.DataFrame*) – Data of shape [n_samples, n_features].

**Returns** Predicted values.

**Return type** pd.Series

**Raises** **MethodPropertyNotFoundError** – If estimator does not have a predict method or a component_obj that implements predict.

**predict_proba**(*self*, *X: pandas.DataFrame*) → pandas.Series

Make probability estimates for labels.

**Parameters** **X** (*pd.DataFrame*) – Features.

**Returns** Probability estimates.

**Return type** pd.Series

**Raises** **MethodPropertyNotFoundError** – If estimator does not have a predict_proba method or a component_obj that implements predict_proba.

**save**(*self*, *file_path*, *pickle_protocol=cloudpickle.DEFAULT_PROTOCOL*)

Saves component at file path.

**Parameters**

- **file_path** (*str*) – Location to save file.
- **pickle_protocol** (*int*) – The pickle data stream format.

**property supported_problem_types**(*cls*)

Problem types this estimator supports.

**update_parameters**(*self*, *update_dict*, *reset_fit=True*)

Updates the parameter dictionary of the component.

**Parameters**

- **update_dict** (*dict*) – A dict of parameters to update.
- **reset_fit** (*bool, optional*) – If True, will set *_is_fitted* to False.

## Package Contents

### Classes Summary

| | |
|---|---|
| *ARIMARegressor* | Autoregressive Integrated Moving Average Model. The three parameters (p, d, q) are the AR order, the degree of differencing, and the MA order. More information here: https://www.statsmodels.org/devel/generated/statsmodels.tsa.arima.model.ARIMA.html. |
| *BaselineClassifier* | Classifier that predicts using the specified strategy. |
| *BaselineRegressor* | Baseline regressor that uses a simple strategy to make predictions. This is useful as a simple baseline regressor to compare with other regressors. |
| *CatBoostClassifier* | CatBoost Classifier, a classifier that uses gradient-boosting on decision trees. CatBoost is an open-source library and natively supports categorical features. |
| *CatBoostRegressor* | CatBoost Regressor, a regressor that uses gradient-boosting on decision trees. CatBoost is an open-source library and natively supports categorical features. |
| *DecisionTreeClassifier* | Decision Tree Classifier. |
| *DecisionTreeRegressor* | Decision Tree Regressor. |
| *ElasticNetClassifier* | Elastic Net Classifier. Uses Logistic Regression with elasticnet penalty as the base estimator. |
| *ElasticNetRegressor* | Elastic Net Regressor. |
| *Estimator* | A component that fits and predicts given data. |
| *ExponentialSmoothingRegressor* | Holt-Winters Exponential Smoothing Forecaster. |
| *ExtraTreesClassifier* | Extra Trees Classifier. |
| *ExtraTreesRegressor* | Extra Trees Regressor. |
| *KNeighborsClassifier* | K-Nearest Neighbors Classifier. |
| *LightGBMClassifier* | LightGBM Classifier. |
| *LightGBMRegressor* | LightGBM Regressor. |
| *LinearRegressor* | Linear Regressor. |
| *LogisticRegressionClassifier* | Logistic Regression Classifier. |
| *ProphetRegressor* | Prophet is a procedure for forecasting time series data based on an additive model where non-linear trends are fit with yearly, weekly, and daily seasonality, plus holiday effects. It works best with time series that have strong seasonal effects and several seasons of historical data. Prophet is robust to missing data and shifts in the trend, and typically handles outliers well. |
| *RandomForestClassifier* | Random Forest Classifier. |
| *RandomForestRegressor* | Random Forest Regressor. |
| *SVMClassifier* | Support Vector Machine Classifier. |
| *SVMRegressor* | Support Vector Machine Regressor. |
| *TimeSeriesBaselineEstimator* | Time series estimator that predicts using the naive forecasting approach. |
| *VowpalWabbitBinaryClassifier* | Vowpal Wabbit Binary Classifier. |
| *VowpalWabbitMulticlassClassifier* | Vowpal Wabbit Multiclass Classifier. |
| *VowpalWabbitRegressor* | Vowpal Wabbit Regressor. |
| *XGBoostClassifier* | XGBoost Classifier. |
| *XGBoostRegressor* | XGBoost Regressor. |

### Contents

**class** evalml.pipelines.components.estimators.**ARIMARegressor**(*time_index: Optional[Hashable] = None, trend: Optional[str] = None, start_p: int = 2, d: int = 0, start_q: int = 2, max_p: int = 5, max_d: int = 2, max_q: int = 5, seasonal: bool = True, sp: int = 1, n_jobs: int = - 1, random_seed: Union[int, float] = 0, maxiter: int = 10, use_covariates: bool = True, **kwargs*)

Autoregressive Integrated Moving Average Model. The three parameters (p, d, q) are the AR order, the degree of differencing, and the MA order. More information here: https://www.statsmodels.org/devel/generated/statsmodels.tsa.arima.model.ARIMA.html.

Currently ARIMARegressor isn't supported via conda install. It's recommended that it be installed via PyPI.

**Parameters**

- **time_index** (*str*) – Specifies the name of the column in X that provides the datetime objects. Defaults to None.
- **trend** (*str*) – Controls the deterministic trend. Options are ['n', 'c', 't', 'ct'] where 'c' is a constant term, 't' indicates a linear trend, and 'ct' is both. Can also be an iterable when defining a polynomial, such as [1, 1, 0, 1].
- **start_p** (*int*) – Minimum Autoregressive order. Defaults to 2.
- **d** (*int*) – Minimum Differencing degree. Defaults to 0.
- **start_q** (*int*) – Minimum Moving Average order. Defaults to 2.
- **max_p** (*int*) – Maximum Autoregressive order. Defaults to 5.
- **max_d** (*int*) – Maximum Differencing degree. Defaults to 2.
- **max_q** (*int*) – Maximum Moving Average order. Defaults to 5.
- **seasonal** (*boolean*) – Whether to fit a seasonal model to ARIMA. Defaults to True.
- **sp** (*int or str*) – Period for seasonal differencing, specifically the number of periods in each season. If "detect", this model will automatically detect this parameter (given the time series is a standard frequency) and will fall back to 1 (no seasonality) if it cannot be detected. Defaults to 1.
- **n_jobs** (*int or None*) – Non-negative integer describing level of parallelism used for pipelines. Defaults to -1.
- **random_seed** (*int*) – Seed for the random number generator. Defaults to 0.

**Attributes**

| | |
|---|---|
| **hyper-parameter_ranges** | { "start_p": Integer(1, 3), "d": Integer(0, 2), "start_q": Integer(1, 3), "max_p": Integer(3, 10), "max_d": Integer(2, 5), "max_q": Integer(3, 10), "seasonal": [True, False],} |
| **max_cols** | 7 |
| **max_rows** | 1000 |
| **model_family** | ModelFamily.ARIMA |
| **modifies_features** | True |
| **modifies_target** | False |
| **name** | ARIMA Regressor |
| **supported_problem_types** | [ProblemTypes.TIME_SERIES_REGRESSION] |
| **training_only** | False |

**Methods**

| | |
|---|---|
| *clone* | Constructs a new component with the same parameters and random state. |
| *default_parameters* | Returns the default parameters for this component. |
| *describe* | Describe a component and its parameters. |
| *feature_importance* | Returns array of 0's with a length of 1 as feature_importance is not defined for ARIMA regressor. |
| *fit* | Fits ARIMA regressor to data. |
| *get_prediction_intervals* | Find the prediction intervals using the fitted ARIMARegressor. |
| *load* | Loads component at file path. |
| *needs_fitting* | Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances. |
| *parameters* | Returns the parameters which were used to initialize the component. |
| *predict* | Make predictions using fitted ARIMA regressor. |
| *predict_proba* | Make probability estimates for labels. |
| *save* | Saves component at file path. |
| *update_parameters* | Updates the parameter dictionary of the component. |

**clone**(*self*)

Constructs a new component with the same parameters and random state.

**Returns** A new instance of this component with identical parameters and random state.

**default_parameters**(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

**Returns** Default parameters for this component.

**Return type** dict

**describe**(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

**Parameters**

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description as dictionary in the format {"name": name, "parameters": parameters}

**Returns** Returns dictionary if return_dict is True, else None.

**Return type** None or dict

**property feature_importance**(*self*) → numpy.ndarray

Returns array of 0's with a length of 1 as feature_importance is not defined for ARIMA regressor.

**fit**(*self, X: pandas.DataFrame, y: Optional[pandas.Series] = None*)

Fits ARIMA regressor to data.

**Parameters**

- **X** (*pd.DataFrame*) – The input training data of shape [n_samples, n_features].
- **y** (*pd.Series*) – The target training data of length [n_samples].

**Returns** self

**Raises** **ValueError** – If y was not passed in.

**get_prediction_intervals**(*self, X: pandas.DataFrame, y: pandas.Series = None, coverage: List[float] = None, predictions: pandas.Series = None*) → Dict[str, pandas.Series]

Find the prediction intervals using the fitted ARIMARegressor.

**Parameters**

- **X** (*pd.DataFrame*) – Data of shape [n_samples, n_features].
- **y** (*pd.Series*) – Target data. Optional.
- **coverage** (*list[float]*) – A list of floats between the values 0 and 1 that the upper and lower bounds of the prediction interval should be calculated for.
- **predictions** (*pd.Series*) – Not used for ARIMA regressor.

**Returns** Prediction intervals, keys are in the format {coverage}_lower or {coverage}_upper.

**Return type** dict

**static load**(*file_path*)

Loads component at file path.

**Parameters** **file_path** (*str*) – Location to load file.

**Returns** ComponentBase object

**needs_fitting**(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need to be fit or whose fit methods do nothing.

**Returns** True.

**property parameters**(*self*)

Returns the parameters which were used to initialize the component.

**predict**(*self*, *X: pandas.DataFrame*, *y: Optional[pandas.Series] = None*) → pandas.Series

Make predictions using fitted ARIMA regressor.

**Parameters**

- **X** (*pd.DataFrame*) – Data of shape [n_samples, n_features].
- **y** (*pd.Series*) – Target data.

**Returns** Predicted values.

**Return type** pd.Series

**Raises** **ValueError** – If X was passed to *fit* but not passed in *predict*.

**predict_proba**(*self*, *X: pandas.DataFrame*) → pandas.Series

Make probability estimates for labels.

**Parameters** **X** (*pd.DataFrame*) – Features.

**Returns** Probability estimates.

**Return type** pd.Series

**Raises** **MethodPropertyNotFoundError** – If estimator does not have a predict_proba method or a component_obj that implements predict_proba.

**save**(*self*, *file_path*, *pickle_protocol=cloudpickle.DEFAULT_PROTOCOL*)

Saves component at file path.

**Parameters**

- **file_path** (*str*) – Location to save file.
- **pickle_protocol** (*int*) – The pickle data stream format.

**update_parameters**(*self*, *update_dict*, *reset_fit=True*)

Updates the parameter dictionary of the component.

**Parameters**

- **update_dict** (*dict*) – A dict of parameters to update.
- **reset_fit** (*bool, optional*) – If True, will set *_is_fitted* to False.

**class** evalml.pipelines.components.estimators.**BaselineClassifier**(*strategy='mode'*, *random_seed=0*, ***kwargs*)

Classifier that predicts using the specified strategy.

This is useful as a simple baseline classifier to compare with other classifiers.

**Parameters**

- **strategy** (*str*) – Method used to predict. Valid options are "mode", "random" and "random_weighted". Defaults to "mode".
- **random_seed** (*int*) – Seed for the random number generator. Defaults to 0.

**Attributes**

| | |
|---|---|
| **hyper-parameter_ranges** | {} |
| **model_family** | ModelFamily.BASELINE |
| **modifies_features** | True |
| **modifies_target** | False |
| **name** | Baseline Classifier |
| **supported_problem_types** | [ProblemTypes.BINARY, ProblemTypes.MULTICLASS] |
| **training_only** | False |

**Methods**

| | |
|---|---|
| *classes_* | Returns class labels. Will return None before fitting. |
| *clone* | Constructs a new component with the same parameters and random state. |
| *default_parameters* | Returns the default parameters for this component. |
| *describe* | Describe a component and its parameters. |
| *feature_importance* | Returns importance associated with each feature. Since baseline classifiers do not use input features to calculate predictions, returns an array of zeroes. |
| *fit* | Fits baseline classifier component to data. |
| *get_prediction_intervals* | Find the prediction intervals using the fitted regressor. |
| *load* | Loads component at file path. |
| *needs_fitting* | Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances. |
| *parameters* | Returns the parameters which were used to initialize the component. |
| *predict* | Make predictions using the baseline classification strategy. |
| *predict_proba* | Make prediction probabilities using the baseline classification strategy. |
| *save* | Saves component at file path. |
| *update_parameters* | Updates the parameter dictionary of the component. |

**property classes_**(*self*)

Returns class labels. Will return None before fitting.

**Returns** Class names

**Return type** list[str] or list(float)

**clone**(*self*)

Constructs a new component with the same parameters and random state.

**Returns** A new instance of this component with identical parameters and random state.

**default_parameters**(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

**Returns** Default parameters for this component.

**Return type** dict

`describe`(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

**Parameters**

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description as dictionary in the format {"name": name, "parameters": parameters}

**Returns** Returns dictionary if return_dict is True, else None.

**Return type** None or dict

`property feature_importance`(*self*)

Returns importance associated with each feature. Since baseline classifiers do not use input features to calculate predictions, returns an array of zeroes.

**Returns** An array of zeroes

**Return type** pd.Series

`fit`(*self*, *X*, *y=None*)

Fits baseline classifier component to data.

**Parameters**

- **X** (*pd.DataFrame*) – The input training data of shape [n_samples, n_features].
- **y** (*pd.Series*) – The target training data of length [n_samples].

**Returns** self

**Raises** **ValueError** – If y is None.

`get_prediction_intervals`(*self*, *X: pandas.DataFrame*, *y: Optional[pandas.Series] = None*, *coverage: List[float] = None*, *predictions: pandas.Series = None*) → Dict[str, pandas.Series]

Find the prediction intervals using the fitted regressor.

This function takes the predictions of the fitted estimator and calculates the rolling standard deviation across all predictions using a window size of 5. The lower and upper predictions are determined by taking the percent point (quantile) function of the lower tail probability at each bound multiplied by the rolling standard deviation.

**Parameters**

- **X** (*pd.DataFrame*) – Data of shape [n_samples, n_features].
- **y** (*pd.Series*) – Target data. Ignored.
- **coverage** (*list[float]*) – A list of floats between the values 0 and 1 that the upper and lower bounds of the prediction interval should be calculated for.
- **predictions** (*pd.Series*) – Optional list of predictions to use. If None, will generate predictions using *X*.

**Returns** Prediction intervals, keys are in the format {coverage}_lower or {coverage}_upper.

**Return type** dict

**Raises** **MethodPropertyNotFoundError** – If the estimator does not support Time Series Regression as a problem type.

**static load**(*file_path*)

Loads component at file path.

**Parameters** **file_path** (*str*) – Location to load file.

**Returns** ComponentBase object

**needs_fitting**(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need to be fit or whose fit methods do nothing.

**Returns** True.

**property parameters**(*self*)

Returns the parameters which were used to initialize the component.

**predict**(*self*, *X*)

Make predictions using the baseline classification strategy.

**Parameters** **X** (*pd.DataFrame*) – Data of shape [n_samples, n_features].

**Returns** Predicted values.

**Return type** pd.Series

**predict_proba**(*self*, *X*)

Make prediction probabilities using the baseline classification strategy.

**Parameters** **X** (*pd.DataFrame*) – Data of shape [n_samples, n_features].

**Returns** Predicted probability values.

**Return type** pd.DataFrame

**save**(*self*, *file_path*, *pickle_protocol=cloudpickle.DEFAULT_PROTOCOL*)

Saves component at file path.

**Parameters**

- **file_path** (*str*) – Location to save file.
- **pickle_protocol** (*int*) – The pickle data stream format.

**update_parameters**(*self*, *update_dict*, *reset_fit=True*)

Updates the parameter dictionary of the component.

**Parameters**

- **update_dict** (*dict*) – A dict of parameters to update.
- **reset_fit** (*bool, optional*) – If True, will set *_is_fitted* to False.

**class** evalml.pipelines.components.estimators.**BaselineRegressor**(*strategy='mean'*, *random_seed=0*, ***kwargs*)

Baseline regressor that uses a simple strategy to make predictions. This is useful as a simple baseline regressor to compare with other regressors.

**Parameters**

- **strategy** (*str*) – Method used to predict. Valid options are "mean", "median". Defaults to "mean".
- **random_seed** (*int*) – Seed for the random number generator. Defaults to 0.

**Attributes**

| | |
|---|---|
| **hyper-parameter_ranges** | {} |
| **model_family** | ModelFamily.BASELINE |
| **modifies_features** | True |
| **modifies_target** | False |
| **name** | Baseline Regressor |
| **supported_problem_types** | [ ProblemTypes.REGRESSION, ProblemTypes.TIME_SERIES_REGRESSION,] |
| **training_only** | False |

**Methods**

| | |
|---|---|
| *clone* | Constructs a new component with the same parameters and random state. |
| *default_parameters* | Returns the default parameters for this component. |
| *describe* | Describe a component and its parameters. |
| *feature_importance* | Returns importance associated with each feature. Since baseline regressors do not use input features to calculate predictions, returns an array of zeroes. |
| *fit* | Fits baseline regression component to data. |
| *get_prediction_intervals* | Find the prediction intervals using the fitted regressor. |
| *load* | Loads component at file path. |
| *needs_fitting* | Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances. |
| *parameters* | Returns the parameters which were used to initialize the component. |
| *predict* | Make predictions using the baseline regression strategy. |
| *predict_proba* | Make probability estimates for labels. |
| *save* | Saves component at file path. |
| *update_parameters* | Updates the parameter dictionary of the component. |

**clone**(*self*)

Constructs a new component with the same parameters and random state.

**Returns** A new instance of this component with identical parameters and random state.

**default_parameters**(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

**Returns** Default parameters for this component.

**Return type** dict

`describe`(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

**Parameters**

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description as dictionary in the format {"name": name, "parameters": parameters}

**Returns** Returns dictionary if return_dict is True, else None.

**Return type** None or dict

`property feature_importance`(*self*)

Returns importance associated with each feature. Since baseline regressors do not use input features to calculate predictions, returns an array of zeroes.

**Returns** An array of zeroes.

**Return type** np.ndarray (float)

`fit`(*self*, *X*, *y=None*)

Fits baseline regression component to data.

**Parameters**

- **X** (*pd.DataFrame*) – The input training data of shape [n_samples, n_features].
- **y** (*pd.Series*) – The target training data of length [n_samples].

**Returns** self

**Raises** **ValueError** – If input y is None.

`get_prediction_intervals`(*self*, *X: pandas.DataFrame*, *y: Optional[pandas.Series] = None*, *coverage: List[float] = None*, *predictions: pandas.Series = None*) → Dict[str, pandas.Series]

Find the prediction intervals using the fitted regressor.

This function takes the predictions of the fitted estimator and calculates the rolling standard deviation across all predictions using a window size of 5. The lower and upper predictions are determined by taking the percent point (quantile) function of the lower tail probability at each bound multiplied by the rolling standard deviation.

**Parameters**

- **X** (*pd.DataFrame*) – Data of shape [n_samples, n_features].
- **y** (*pd.Series*) – Target data. Ignored.
- **coverage** (*list[float]*) – A list of floats between the values 0 and 1 that the upper and lower bounds of the prediction interval should be calculated for.
- **predictions** (*pd.Series*) – Optional list of predictions to use. If None, will generate predictions using *X*.

**Returns** Prediction intervals, keys are in the format {coverage}_lower or {coverage}_upper.

**Return type** dict

**Raises** `MethodPropertyNotFoundError` – If the estimator does not support Time Series Regression as a problem type.

`static load`(*file_path*)

Loads component at file path.

**Parameters** `file_path` (*str*) – Location to load file.

**Returns** ComponentBase object

`needs_fitting`(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need to be fit or whose fit methods do nothing.

**Returns** True.

`property parameters`(*self*)

Returns the parameters which were used to initialize the component.

`predict`(*self*, *X*)

Make predictions using the baseline regression strategy.

**Parameters** `X` (*pd.DataFrame*) – Data of shape [n_samples, n_features].

**Returns** Predicted values.

**Return type** pd.Series

`predict_proba`(*self*, *X: pandas.DataFrame*) → pandas.Series

Make probability estimates for labels.

**Parameters** `X` (*pd.DataFrame*) – Features.

**Returns** Probability estimates.

**Return type** pd.Series

**Raises** `MethodPropertyNotFoundError` – If estimator does not have a predict_proba method or a component_obj that implements predict_proba.

`save`(*self*, *file_path*, *pickle_protocol=cloudpickle.DEFAULT_PROTOCOL*)

Saves component at file path.

**Parameters**

- `file_path` (*str*) – Location to save file.
- `pickle_protocol` (*int*) – The pickle data stream format.

`update_parameters`(*self*, *update_dict*, *reset_fit=True*)

Updates the parameter dictionary of the component.

**Parameters**

- `update_dict` (*dict*) – A dict of parameters to update.
- `reset_fit` (*bool, optional*) – If True, will set *_is_fitted* to False.

**class** evalml.pipelines.components.estimators.**CatBoostClassifier**(*n_estimators=10*, *eta=0.03*, *max_depth=6*, *bootstrap_type=None*, *silent=True*, *allow_writing_files=False*, *random_seed=0*, *n_jobs=- 1*, ***kwargs*)

CatBoost Classifier, a classifier that uses gradient-boosting on decision trees. CatBoost is an open-source library and natively supports categorical features.

For more information, check out https://catboost.ai/

**Parameters**

- **n_estimators** (*float*) – The maximum number of trees to build. Defaults to 10.
- **eta** (*float*) – The learning rate. Defaults to 0.03.
- **max_depth** (*int*) – The maximum tree depth for base learners. Defaults to 6.
- **bootstrap_type** (*string*) – Defines the method for sampling the weights of objects. Available methods are 'Bayesian', 'Bernoulli', 'MVS'. Defaults to None.
- **silent** (*boolean*) – Whether to use the "silent" logging mode. Defaults to True.
- **allow_writing_files** (*boolean*) – Whether to allow writing snapshot files while training. Defaults to False.
- **n_jobs** (*int or None*) – Number of jobs to run in parallel. -1 uses all processes. Defaults to -1.
- **random_seed** (*int*) – Seed for the random number generator. Defaults to 0.

**Attributes**

| | |
|---|---|
| **hyperparameter_ranges** | { "n_estimators": Integer(4, 100), "eta": Real(0.000001, 1), "max_depth": Integer(4, 10),} |
| **model_family** | ModelFamily.CATBOOST |
| **modifies_features** | True |
| **modifies_target** | False |
| **name** | CatBoost Classifier |
| **supported_problem_types** | [ ProblemTypes.BINARY, ProblemTypes.MULTICLASS, ProblemTypes.TIME_SERIES_BINARY, ProblemTypes.TIME_SERIES_MULTICLASS,] |
| **training_only** | False |

**Methods**

| | |
|---|---|
| *clone* | Constructs a new component with the same parameters and random state. |
| *default_parameters* | Returns the default parameters for this component. |
| *describe* | Describe a component and its parameters. |
| *feature_importance* | Feature importance of fitted CatBoost classifier. |
| *fit* | Fits CatBoost classifier component to data. |
| *get_prediction_intervals* | Find the prediction intervals using the fitted regressor. |
| *load* | Loads component at file path. |
| *needs_fitting* | Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances. |
| *parameters* | Returns the parameters which were used to initialize the component. |
| *predict* | Make predictions using the fitted CatBoost classifier. |
| *predict_proba* | Make prediction probabilities using the fitted CatBoost classifier. |
| *save* | Saves component at file path. |
| *update_parameters* | Updates the parameter dictionary of the component. |

**clone**(*self*)

Constructs a new component with the same parameters and random state.

**Returns** A new instance of this component with identical parameters and random state.

**default_parameters**(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

**Returns** Default parameters for this component.

**Return type** dict

**describe**(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

**Parameters**

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description as dictionary in the format {"name": name, "parameters": parameters}

**Returns** Returns dictionary if return_dict is True, else None.

**Return type** None or dict

**property feature_importance**(*self*)

Feature importance of fitted CatBoost classifier.

**fit**(*self*, *X*, *y=None*)

Fits CatBoost classifier component to data.

**Parameters**

- **X** (*pd.DataFrame*) – The input training data of shape [n_samples, n_features].
- **y** (*pd.Series*) – The target training data of length [n_samples].

**Returns** self

**get_prediction_intervals**(*self*, *X: pandas.DataFrame*, *y: Optional[pandas.Series] = None*, *coverage: List[float] = None*, *predictions: pandas.Series = None*) → Dict[str, pandas.Series]

Find the prediction intervals using the fitted regressor.

This function takes the predictions of the fitted estimator and calculates the rolling standard deviation across all predictions using a window size of 5. The lower and upper predictions are determined by taking the percent point (quantile) function of the lower tail probability at each bound multiplied by the rolling standard deviation.

**Parameters**

- **X** (*pd.DataFrame*) – Data of shape [n_samples, n_features].
- **y** (*pd.Series*) – Target data. Ignored.
- **coverage** (*list[float]*) – A list of floats between the values 0 and 1 that the upper and lower bounds of the prediction interval should be calculated for.
- **predictions** (*pd.Series*) – Optional list of predictions to use. If None, will generate predictions using *X*.

**Returns** Prediction intervals, keys are in the format {coverage}_lower or {coverage}_upper.

**Return type** dict

**Raises** **MethodPropertyNotFoundError** – If the estimator does not support Time Series Regression as a problem type.

**static load**(*file_path*)

Loads component at file path.

**Parameters** **file_path** (*str*) – Location to load file.

**Returns** ComponentBase object

**needs_fitting**(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need to be fit or whose fit methods do nothing.

**Returns** True.

**property parameters**(*self*)

Returns the parameters which were used to initialize the component.

**predict**(*self*, *X*)

Make predictions using the fitted CatBoost classifier.

**Parameters** **X** (*pd.DataFrame*) – Data of shape [n_samples, n_features].

**Returns** Predicted values.

**Return type** pd.Series

**predict_proba**(*self*, *X*)

Make prediction probabilities using the fitted CatBoost classifier.

**Parameters** **X** (*pd.DataFrame*) – Data of shape [n_samples, n_features].

**Returns** Predicted probability values.

**Return type** pd.DataFrame

**save**(*self*, *file_path*, *pickle_protocol=cloudpickle.DEFAULT_PROTOCOL*)

Saves component at file path.

**Parameters**

- **file_path** (*str*) – Location to save file.
- **pickle_protocol** (*int*) – The pickle data stream format.

**update_parameters**(*self*, *update_dict*, *reset_fit=True*)

Updates the parameter dictionary of the component.

**Parameters**

- **update_dict** (*dict*) – A dict of parameters to update.
- **reset_fit** (*bool, optional*) – If True, will set *_is_fitted* to False.

**class** evalml.pipelines.components.estimators.**CatBoostRegressor**(*n_estimators=10*, *eta=0.03*, *max_depth=6*, *bootstrap_type=None*, *silent=False*, *allow_writing_files=False*, *random_seed=0*, *n_jobs=- 1*, ***kwargs*)

CatBoost Regressor, a regressor that uses gradient-boosting on decision trees. CatBoost is an open-source library and natively supports categorical features.

For more information, check out https://catboost.ai/

**Parameters**

- **n_estimators** (*float*) – The maximum number of trees to build. Defaults to 10.
- **eta** (*float*) – The learning rate. Defaults to 0.03.
- **max_depth** (*int*) – The maximum tree depth for base learners. Defaults to 6.
- **bootstrap_type** (*string*) – Defines the method for sampling the weights of objects. Available methods are 'Bayesian', 'Bernoulli', 'MVS'. Defaults to None.
- **silent** (*boolean*) – Whether to use the "silent" logging mode. Defaults to True.
- **allow_writing_files** (*boolean*) – Whether to allow writing snapshot files while training. Defaults to False.
- **n_jobs** (*int or None*) – Number of jobs to run in parallel. -1 uses all processes. Defaults to -1.
- **random_seed** (*int*) – Seed for the random number generator. Defaults to 0.

**Attributes**

| | |
|---|---|
| **hyper-parameter_ranges** | { "n_estimators": Integer(4, 100), "eta": Real(0.000001, 1), "max_depth": Integer(4, 10),} |
| **model_family** | ModelFamily.CATBOOST |
| **modifies_features** | True |
| **modifies_target** | False |
| **name** | CatBoost Regressor |
| **supported_problem_types** | [ ProblemTypes.REGRESSION, ProblemTypes.TIME_SERIES_REGRESSION,] |
| **training_only** | False |

**Methods**

| | |
|---|---|
| *clone* | Constructs a new component with the same parameters and random state. |
| *default_parameters* | Returns the default parameters for this component. |
| *describe* | Describe a component and its parameters. |
| *feature_importance* | Feature importance of fitted CatBoost regressor. |
| *fit* | Fits CatBoost regressor component to data. |
| *get_prediction_intervals* | Find the prediction intervals using the fitted regressor. |
| *load* | Loads component at file path. |
| *needs_fitting* | Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances. |
| *parameters* | Returns the parameters which were used to initialize the component. |
| *predict* | Make predictions using the fitted CatBoost regressor. |
| *predict_proba* | Make probability estimates for labels. |
| *save* | Saves component at file path. |
| *update_parameters* | Updates the parameter dictionary of the component. |

**clone**(*self*)

Constructs a new component with the same parameters and random state.

**Returns** A new instance of this component with identical parameters and random state.

**default_parameters**(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

**Returns** Default parameters for this component.

**Return type** dict

**describe**(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

**Parameters**

- **print_name** (*bool, optional*) – whether to print name of component

- **return_dict** (*bool, optional*) – whether to return description as dictionary in the format {"name": name, "parameters": parameters}

**Returns** Returns dictionary if return_dict is True, else None.

**Return type** None or dict

**property feature_importance**(*self*)

Feature importance of fitted CatBoost regressor.

**fit**(*self*, *X*, *y=None*)

Fits CatBoost regressor component to data.

**Parameters**

- **X** (*pd.DataFrame*) – The input training data of shape [n_samples, n_features].
- **y** (*pd.Series*) – The target training data of length [n_samples].

**Returns** self

**get_prediction_intervals**(*self*, *X: pandas.DataFrame*, *y: Optional[pandas.Series] = None*, *coverage: List[float] = None*, *predictions: pandas.Series = None*) → Dict[str, pandas.Series]

Find the prediction intervals using the fitted regressor.

This function takes the predictions of the fitted estimator and calculates the rolling standard deviation across all predictions using a window size of 5. The lower and upper predictions are determined by taking the percent point (quantile) function of the lower tail probability at each bound multiplied by the rolling standard deviation.

**Parameters**

- **X** (*pd.DataFrame*) – Data of shape [n_samples, n_features].
- **y** (*pd.Series*) – Target data. Ignored.
- **coverage** (*list[float]*) – A list of floats between the values 0 and 1 that the upper and lower bounds of the prediction interval should be calculated for.
- **predictions** (*pd.Series*) – Optional list of predictions to use. If None, will generate predictions using *X*.

**Returns** Prediction intervals, keys are in the format {coverage}_lower or {coverage}_upper.

**Return type** dict

**Raises** **MethodPropertyNotFoundError** – If the estimator does not support Time Series Regression as a problem type.

**static load**(*file_path*)

Loads component at file path.

**Parameters** **file_path** (*str*) – Location to load file.

**Returns** ComponentBase object

**needs_fitting**(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need to be fit or whose fit methods do nothing.

**Returns** True.

**property parameters**(*self*)

Returns the parameters which were used to initialize the component.

**predict**(*self*, *X*)

Make predictions using the fitted CatBoost regressor.

**Parameters** **X** (*pd.DataFrame*) – Data of shape [n_samples, n_features].

**Returns** Predicted values.

**Return type** pd.DataFrame

**predict_proba**(*self*, *X: pandas.DataFrame*) → pandas.Series

Make probability estimates for labels.

**Parameters** **X** (*pd.DataFrame*) – Features.

**Returns** Probability estimates.

**Return type** pd.Series

**Raises** **MethodPropertyNotFoundError** – If estimator does not have a predict_proba method or a component_obj that implements predict_proba.

**save**(*self*, *file_path*, *pickle_protocol=cloudpickle.DEFAULT_PROTOCOL*)

Saves component at file path.

**Parameters**

- **file_path** (*str*) – Location to save file.
- **pickle_protocol** (*int*) – The pickle data stream format.

**update_parameters**(*self*, *update_dict*, *reset_fit=True*)

Updates the parameter dictionary of the component.

**Parameters**

- **update_dict** (*dict*) – A dict of parameters to update.
- **reset_fit** (*bool, optional*) – If True, will set *_is_fitted* to False.

**class** evalml.pipelines.components.estimators.**DecisionTreeClassifier**(*criterion='gini'*, *max_features='auto'*, *max_depth=6*, *min_samples_split=2*, *min_weight_fraction_leaf=0.0*, *random_seed=0*, ***kwargs*)

Decision Tree Classifier.

**Parameters**

- **criterion** (*{"gini", "entropy"}*) – The function to measure the quality of a split. Supported criteria are "gini" for the Gini impurity and "entropy" for the information gain. Defaults to "gini".
- **max_features** (*int, float or {"auto", "sqrt", "log2"}*) – The number of features to consider when looking for the best split:
  - If int, then consider max_features features at each split.
  - If float, then max_features is a fraction and int(max_features * n_features) features are considered at each split.

- If "auto", then max_features=sqrt(n_features).
- If "sqrt", then max_features=sqrt(n_features).
- If "log2", then max_features=log2(n_features).
- If None, then max_features = n_features.

The search for a split does not stop until at least one valid partition of the node samples is found, even if it requires to effectively inspect more than max_features features. Defaults to "auto".

- **max_depth** (*int*) – The maximum depth of the tree. Defaults to 6.
- **min_samples_split** (*int or float*) – The minimum number of samples required to split an internal node:
  - If int, then consider min_samples_split as the minimum number.
  - If float, then min_samples_split is a fraction and ceil(min_samples_split * n_samples) are the minimum number of samples for each split.

  Defaults to 2.
- **min_weight_fraction_leaf** (*float*) – The minimum weighted fraction of the sum total of weights (of all the input samples) required to be at a leaf node. Defaults to 0.0.
- **random_seed** (*int*) – Seed for the random number generator. Defaults to 0.

**Attributes**

| | |
|---|---|
| **hyper-parameter_ranges** | { "criterion": ["gini", "entropy"], "max_features": ["auto", "sqrt", "log2"], "max_depth": Integer(4, 10),} |
| **model_family** | ModelFamily.DECISION_TREE |
| **modifies_features** | True |
| **modifies_target** | False |
| **name** | Decision Tree Classifier |
| **supported_problem_types** | [ ProblemTypes.BINARY, ProblemTypes.MULTICLASS, ProblemTypes.TIME_SERIES_BINARY, ProblemTypes.TIME_SERIES_MULTICLASS,] |
| **training_only** | False |

**Methods**

| | |
|---|---|
| *clone* | Constructs a new component with the same parameters and random state. |
| *default_parameters* | Returns the default parameters for this component. |
| *describe* | Describe a component and its parameters. |
| *feature_importance* | Returns importance associated with each feature. |
| *fit* | Fits estimator to data. |
| *get_prediction_intervals* | Find the prediction intervals using the fitted regressor. |
| *load* | Loads component at file path. |
| *needs_fitting* | Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances. |
| *parameters* | Returns the parameters which were used to initialize the component. |
| *predict* | Make predictions using selected features. |
| *predict_proba* | Make probability estimates for labels. |
| *save* | Saves component at file path. |
| *update_parameters* | Updates the parameter dictionary of the component. |

**clone**(*self*)

Constructs a new component with the same parameters and random state.

**Returns** A new instance of this component with identical parameters and random state.

**default_parameters**(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

**Returns** Default parameters for this component.

**Return type** dict

**describe**(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

**Parameters**

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description as dictionary in the format {"name": name, "parameters": parameters}

**Returns** Returns dictionary if return_dict is True, else None.

**Return type** None or dict

**property feature_importance**(*self*) → pandas.Series

Returns importance associated with each feature.

**Returns** Importance associated with each feature.

**Return type** np.ndarray

**Raises** **MethodPropertyNotFoundError** – If estimator does not have a feature_importance method or a component_obj that implements feature_importance.

**fit**(*self*, *X: pandas.DataFrame*, *y: Optional[pandas.Series] = None*)

Fits estimator to data.

**Parameters**

- **X** (*pd.DataFrame*) – The input training data of shape [n_samples, n_features].
- **y** (*pd.Series, optional*) – The target training data of length [n_samples].

**Returns** self

**get_prediction_intervals**(*self, X: pandas.DataFrame, y: Optional[pandas.Series] = None, coverage: List[float] = None, predictions: pandas.Series = None*) → Dict[str, pandas.Series]

Find the prediction intervals using the fitted regressor.

This function takes the predictions of the fitted estimator and calculates the rolling standard deviation across all predictions using a window size of 5. The lower and upper predictions are determined by taking the percent point (quantile) function of the lower tail probability at each bound multiplied by the rolling standard deviation.

**Parameters**

- **X** (*pd.DataFrame*) – Data of shape [n_samples, n_features].
- **y** (*pd.Series*) – Target data. Ignored.
- **coverage** (*list[float]*) – A list of floats between the values 0 and 1 that the upper and lower bounds of the prediction interval should be calculated for.
- **predictions** (*pd.Series*) – Optional list of predictions to use. If None, will generate predictions using *X*.

**Returns** Prediction intervals, keys are in the format {coverage}_lower or {coverage}_upper.

**Return type** dict

**Raises** **MethodPropertyNotFoundError** – If the estimator does not support Time Series Regression as a problem type.

**static load**(*file_path*)

Loads component at file path.

**Parameters** **file_path** (*str*) – Location to load file.

**Returns** ComponentBase object

**needs_fitting**(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need to be fit or whose fit methods do nothing.

**Returns** True.

**property parameters**(*self*)

Returns the parameters which were used to initialize the component.

**predict**(*self, X: pandas.DataFrame*) → pandas.Series

Make predictions using selected features.

**Parameters** **X** (*pd.DataFrame*) – Data of shape [n_samples, n_features].

**Returns** Predicted values.

**Return type** pd.Series

**Raises** `MethodPropertyNotFoundError` – If estimator does not have a predict method or a component_obj that implements predict.

`predict_proba`(*self*, *X: pandas.DataFrame*) → pandas.Series

Make probability estimates for labels.

**Parameters** `X` (*pd.DataFrame*) – Features.

**Returns** Probability estimates.

**Return type** pd.Series

**Raises** `MethodPropertyNotFoundError` – If estimator does not have a predict_proba method or a component_obj that implements predict_proba.

`save`(*self*, *file_path*, *pickle_protocol=cloudpickle.DEFAULT_PROTOCOL*)

Saves component at file path.

**Parameters**

- `file_path` (*str*) – Location to save file.
- `pickle_protocol` (*int*) – The pickle data stream format.

`update_parameters`(*self*, *update_dict*, *reset_fit=True*)

Updates the parameter dictionary of the component.

**Parameters**

- `update_dict` (*dict*) – A dict of parameters to update.
- `reset_fit` (*bool, optional*) – If True, will set *_is_fitted* to False.

**class** `evalml.pipelines.components.estimators.`**`DecisionTreeRegressor`**(*criterion='squared_error'*, *max_features='auto'*, *max_depth=6*, *min_samples_split=2*, *min_weight_fraction_leaf=0.0*, *random_seed=0*, ***kwargs*)

Decision Tree Regressor.

**Parameters**

- `criterion` (*{"squared_error", "friedman_mse", "absolute_error", "poisson"}*) – The function to measure the quality of a split. Supported criteria are:
  - "squared_error" for the mean squared error, which is equal to variance reduction as feature selection criterion and minimizes the L2 loss using the mean of each terminal node
  - "friedman_mse", which uses mean squared error with Friedman"s improvement score for potential splits
  - "absolute_error" for the mean absolute error, which minimizes the L1 loss using the median of each terminal node,
  - "poisson" which uses reduction in Poisson deviance to find splits.
- `max_features` (*int, float or {"auto", "sqrt", "log2"}*) – The number of features to consider when looking for the best split:
  - If int, then consider max_features features at each split.

- If float, then max_features is a fraction and int(max_features * n_features) features are considered at each split.
- If "auto", then max_features=sqrt(n_features).
- If "sqrt", then max_features=sqrt(n_features).
- If "log2", then max_features=log2(n_features).
- If None, then max_features = n_features.

The search for a split does not stop until at least one valid partition of the node samples is found, even if it requires to effectively inspect more than max_features features.

- **max_depth** (*int*) – The maximum depth of the tree. Defaults to 6.
- **min_samples_split** (*int or float*) – The minimum number of samples required to split an internal node:
  - If int, then consider min_samples_split as the minimum number.
  - If float, then min_samples_split is a fraction and ceil(min_samples_split * n_samples) are the minimum number of samples for each split.

  Defaults to 2.
- **min_weight_fraction_leaf** (*float*) – The minimum weighted fraction of the sum total of weights (of all the input samples) required to be at a leaf node. Defaults to 0.0.
- **random_seed** (*int*) – Seed for the random number generator. Defaults to 0.

**Attributes**

| | |
|---|---|
| **hyper-parameter_ranges** | { "criterion": ["squared_error", "friedman_mse", "absolute_error"], "max_features": ["auto", "sqrt", "log2"], "max_depth": Integer(4, 10),} |
| **model_family** | ModelFamily.DECISION_TREE |
| **modifies_features** | True |
| **modifies_target** | False |
| **name** | Decision Tree Regressor |
| **supported_problem_types** | [ ProblemTypes.REGRESSION, ProblemTypes.TIME_SERIES_REGRESSION,] |
| **training_only** | False |

**Methods**

| | |
|---|---|
| *clone* | Constructs a new component with the same parameters and random state. |
| *default_parameters* | Returns the default parameters for this component. |
| *describe* | Describe a component and its parameters. |
| *feature_importance* | Returns importance associated with each feature. |
| *fit* | Fits estimator to data. |
| *get_prediction_intervals* | Find the prediction intervals using the fitted regressor. |
| *load* | Loads component at file path. |
| *needs_fitting* | Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances. |
| *parameters* | Returns the parameters which were used to initialize the component. |
| *predict* | Make predictions using selected features. |
| *predict_proba* | Make probability estimates for labels. |
| *save* | Saves component at file path. |
| *update_parameters* | Updates the parameter dictionary of the component. |

**clone**(*self*)

Constructs a new component with the same parameters and random state.

**Returns** A new instance of this component with identical parameters and random state.

**default_parameters**(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

**Returns** Default parameters for this component.

**Return type** dict

**describe**(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

**Parameters**

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description as dictionary in the format {"name": name, "parameters": parameters}

**Returns** Returns dictionary if return_dict is True, else None.

**Return type** None or dict

**property feature_importance**(*self*) → pandas.Series

Returns importance associated with each feature.

**Returns** Importance associated with each feature.

**Return type** np.ndarray

**Raises** **MethodPropertyNotFoundError** – If estimator does not have a feature_importance method or a component_obj that implements feature_importance.

**fit**(*self*, *X: pandas.DataFrame*, *y: Optional[pandas.Series] = None*)

Fits estimator to data.

**Parameters**

- **X** (*pd.DataFrame*) – The input training data of shape [n_samples, n_features].
- **y** (*pd.Series, optional*) – The target training data of length [n_samples].

**Returns** self

**get_prediction_intervals**(*self, X: pandas.DataFrame, y: Optional[pandas.Series] = None, coverage: List[float] = None, predictions: pandas.Series = None*) → Dict[str, pandas.Series]

Find the prediction intervals using the fitted regressor.

This function takes the predictions of the fitted estimator and calculates the rolling standard deviation across all predictions using a window size of 5. The lower and upper predictions are determined by taking the percent point (quantile) function of the lower tail probability at each bound multiplied by the rolling standard deviation.

**Parameters**

- **X** (*pd.DataFrame*) – Data of shape [n_samples, n_features].
- **y** (*pd.Series*) – Target data. Ignored.
- **coverage** (*list[float]*) – A list of floats between the values 0 and 1 that the upper and lower bounds of the prediction interval should be calculated for.
- **predictions** (*pd.Series*) – Optional list of predictions to use. If None, will generate predictions using *X*.

**Returns** Prediction intervals, keys are in the format {coverage}_lower or {coverage}_upper.

**Return type** dict

**Raises** **MethodPropertyNotFoundError** – If the estimator does not support Time Series Regression as a problem type.

**static load**(*file_path*)

Loads component at file path.

**Parameters** **file_path** (*str*) – Location to load file.

**Returns** ComponentBase object

**needs_fitting**(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need to be fit or whose fit methods do nothing.

**Returns** True.

**property parameters**(*self*)

Returns the parameters which were used to initialize the component.

**predict**(*self, X: pandas.DataFrame*) → pandas.Series

Make predictions using selected features.

**Parameters** **X** (*pd.DataFrame*) – Data of shape [n_samples, n_features].

**Returns** Predicted values.

**Return type** pd.Series

**Raises** **MethodPropertyNotFoundError** – If estimator does not have a predict method or a component_obj that implements predict.

**predict_proba**(*self*, *X: pandas.DataFrame*) → pandas.Series

Make probability estimates for labels.

**Parameters** **X** (*pd.DataFrame*) – Features.

**Returns** Probability estimates.

**Return type** pd.Series

**Raises** **MethodPropertyNotFoundError** – If estimator does not have a predict_proba method or a component_obj that implements predict_proba.

**save**(*self*, *file_path*, *pickle_protocol=cloudpickle.DEFAULT_PROTOCOL*)

Saves component at file path.

**Parameters**

- **file_path** (*str*) – Location to save file.
- **pickle_protocol** (*int*) – The pickle data stream format.

**update_parameters**(*self*, *update_dict*, *reset_fit=True*)

Updates the parameter dictionary of the component.

**Parameters**

- **update_dict** (*dict*) – A dict of parameters to update.
- **reset_fit** (*bool, optional*) – If True, will set *_is_fitted* to False.

**class** evalml.pipelines.components.estimators.**ElasticNetClassifier**(*penalty='elasticnet'*, *C=1.0*, *l1_ratio=0.15*, *multi_class='auto'*, *solver='saga'*, *n_jobs=- 1*, *random_seed=0*, ***kwargs*)

Elastic Net Classifier. Uses Logistic Regression with elasticnet penalty as the base estimator.

**Parameters**

- **penalty** (*{"l1", "l2", "elasticnet", "none"}*) – The norm used in penalization. Defaults to "elasticnet".
- **C** (*float*) – Inverse of regularization strength. Must be a positive float. Defaults to 1.0.
- **l1_ratio** (*float*) – The mixing parameter, with 0 <= l1_ratio <= 1. Only used if penalty='elasticnet'. Setting l1_ratio=0 is equivalent to using penalty='l2', while setting l1_ratio=1 is equivalent to using penalty='l1'. For 0 < l1_ratio <1, the penalty is a combination of L1 and L2. Defaults to 0.15.
- **multi_class** (*{"auto", "ovr", "multinomial"}*) – If the option chosen is "ovr", then a binary problem is fit for each label. For "multinomial" the loss minimised is the multinomial loss fit across the entire probability distribution, even when the data is binary. "multinomial" is unavailable when solver="liblinear". "auto" selects "ovr" if the data is binary, or if solver="liblinear", and otherwise selects "multinomial". Defaults to "auto".
- **solver** (*{"newton-cg", "lbfgs", "liblinear", "sag", "saga"}*) – Algorithm to use in the optimization problem. For small datasets, "liblinear" is a good choice, whereas "sag" and "saga" are faster for large ones. For multiclass problems, only "newton-cg", "sag", "saga" and "lbfgs" handle multinomial loss; "liblinear" is limited to one-versus-rest schemes.

- "newton-cg", "lbfgs", "sag" and "saga" handle L2 or no penalty
- "liblinear" and "saga" also handle L1 penalty
- "saga" also supports "elasticnet" penalty
- "liblinear" does not support setting penalty='none'

Defaults to "saga".

- **n_jobs** (*int*) – Number of parallel threads used to run xgboost. Note that creating thread contention will significantly slow down the algorithm. Defaults to -1.
- **random_seed** (*int*) – Seed for the random number generator. Defaults to 0.

**Attributes**

| | |
|---|---|
| **hyper-parameter_ranges** | { "C": Real(0.01, 10), "l1_ratio": Real(0, 1)} |
| **model_family** | ModelFamily.LINEAR_MODEL |
| **modifies_features** | True |
| **modifies_target** | False |
| **name** | Elastic Net Classifier |
| **supported_problem_types** | [ ProblemTypes.BINARY, ProblemTypes.MULTICLASS, ProblemTypes.TIME_SERIES_BINARY, ProblemTypes.TIME_SERIES_MULTICLASS,] |
| **training_only** | False |

**Methods**

| | |
|---|---|
| *clone* | Constructs a new component with the same parameters and random state. |
| *default_parameters* | Returns the default parameters for this component. |
| *describe* | Describe a component and its parameters. |
| *feature_importance* | Feature importance for fitted ElasticNet classifier. |
| *fit* | Fits ElasticNet classifier component to data. |
| *get_prediction_intervals* | Find the prediction intervals using the fitted regressor. |
| *load* | Loads component at file path. |
| *needs_fitting* | Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances. |
| *parameters* | Returns the parameters which were used to initialize the component. |
| *predict* | Make predictions using selected features. |
| *predict_proba* | Make probability estimates for labels. |
| *save* | Saves component at file path. |
| *update_parameters* | Updates the parameter dictionary of the component. |

**clone**(*self*)

Constructs a new component with the same parameters and random state.

**Returns** A new instance of this component with identical parameters and random state.

**default_parameters**(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

**Returns** Default parameters for this component.

**Return type** dict

**describe**(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

**Parameters**

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description as dictionary in the format {"name": name, "parameters": parameters}

**Returns** Returns dictionary if return_dict is True, else None.

**Return type** None or dict

**property feature_importance**(*self*)

Feature importance for fitted ElasticNet classifier.

**fit**(*self*, *X*, *y*)

Fits ElasticNet classifier component to data.

**Parameters**

- **X** (*pd.DataFrame*) – The input training data of shape [n_samples, n_features].
- **y** (*pd.Series*) – The target training data of length [n_samples].

**Returns** self

**get_prediction_intervals**(*self*, *X: pandas.DataFrame*, *y: Optional[pandas.Series] = None*, *coverage: List[float] = None*, *predictions: pandas.Series = None*) → Dict[str, pandas.Series]

Find the prediction intervals using the fitted regressor.

This function takes the predictions of the fitted estimator and calculates the rolling standard deviation across all predictions using a window size of 5. The lower and upper predictions are determined by taking the percent point (quantile) function of the lower tail probability at each bound multiplied by the rolling standard deviation.

**Parameters**

- **X** (*pd.DataFrame*) – Data of shape [n_samples, n_features].
- **y** (*pd.Series*) – Target data. Ignored.
- **coverage** (*list[float]*) – A list of floats between the values 0 and 1 that the upper and lower bounds of the prediction interval should be calculated for.
- **predictions** (*pd.Series*) – Optional list of predictions to use. If None, will generate predictions using *X*.

**Returns** Prediction intervals, keys are in the format {coverage}_lower or {coverage}_upper.

**Return type** dict

**Raises** **MethodPropertyNotFoundError** – If the estimator does not support Time Series Regression as a problem type.

**static load**(*file_path*)

Loads component at file path.

**Parameters** **file_path** (*str*) – Location to load file.

**Returns** ComponentBase object

**needs_fitting**(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need to be fit or whose fit methods do nothing.

**Returns** True.

**property parameters**(*self*)

Returns the parameters which were used to initialize the component.

**predict**(*self*, *X: pandas.DataFrame*) → pandas.Series

Make predictions using selected features.

**Parameters** **X** (*pd.DataFrame*) – Data of shape [n_samples, n_features].

**Returns** Predicted values.

**Return type** pd.Series

**Raises** **MethodPropertyNotFoundError** – If estimator does not have a predict method or a component_obj that implements predict.

**predict_proba**(*self*, *X: pandas.DataFrame*) → pandas.Series

Make probability estimates for labels.

**Parameters** **X** (*pd.DataFrame*) – Features.

**Returns** Probability estimates.

**Return type** pd.Series

**Raises** **MethodPropertyNotFoundError** – If estimator does not have a predict_proba method or a component_obj that implements predict_proba.

**save**(*self*, *file_path*, *pickle_protocol=cloudpickle.DEFAULT_PROTOCOL*)

Saves component at file path.

**Parameters**

- **file_path** (*str*) – Location to save file.
- **pickle_protocol** (*int*) – The pickle data stream format.

**update_parameters**(*self*, *update_dict*, *reset_fit=True*)

Updates the parameter dictionary of the component.

**Parameters**

- **update_dict** (*dict*) – A dict of parameters to update.
- **reset_fit** (*bool, optional*) – If True, will set *_is_fitted* to False.

**class** evalml.pipelines.components.estimators.**ElasticNetRegressor**(*alpha=0.0001*, *l1_ratio=0.15*, *max_iter=1000*, *random_seed=0*, ***kwargs*)

Elastic Net Regressor.

**Parameters**

- **alpha** (*float*) – Constant that multiplies the penalty terms. Defaults to 0.0001.
- **l1_ratio** (*float*) – The mixing parameter, with 0 <= l1_ratio <= 1. Only used if penalty='elasticnet'. Setting l1_ratio=0 is equivalent to using penalty='l2', while setting l1_ratio=1 is equivalent to using penalty='l1'. For 0 < l1_ratio <1, the penalty is a combination of L1 and L2. Defaults to 0.15.
- **max_iter** (*int*) – The maximum number of iterations. Defaults to 1000.
- **random_seed** (*int*) – Seed for the random number generator. Defaults to 0.

**Attributes**

| | |
|---|---|
| **hyper-parameter_ranges** | { "alpha": Real(0, 1), "l1_ratio": Real(0, 1),} |
| **model_family** | ModelFamily.LINEAR_MODEL |
| **modifies_features** | True |
| **modifies_target** | False |
| **name** | Elastic Net Regressor |
| **supported_problem_types** | [ ProblemTypes.REGRESSION, ProblemTypes.TIME_SERIES_REGRESSION,] |
| **training_only** | False |

**Methods**

| | |
|---|---|
| *clone* | Constructs a new component with the same parameters and random state. |
| *default_parameters* | Returns the default parameters for this component. |
| *describe* | Describe a component and its parameters. |
| *feature_importance* | Feature importance for fitted ElasticNet regressor. |
| *fit* | Fits estimator to data. |
| *get_prediction_intervals* | Find the prediction intervals using the fitted regressor. |
| *load* | Loads component at file path. |
| *needs_fitting* | Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances. |
| *parameters* | Returns the parameters which were used to initialize the component. |
| *predict* | Make predictions using selected features. |
| *predict_proba* | Make probability estimates for labels. |
| *save* | Saves component at file path. |
| *update_parameters* | Updates the parameter dictionary of the component. |

**clone**(*self*)

Constructs a new component with the same parameters and random state.

**Returns** A new instance of this component with identical parameters and random state.

**default_parameters**(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

**Returns** Default parameters for this component.

**Return type** dict

**describe**(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

**Parameters**

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description as dictionary in the format {"name": name, "parameters": parameters}

**Returns** Returns dictionary if return_dict is True, else None.

**Return type** None or dict

**property feature_importance**(*self*)

Feature importance for fitted ElasticNet regressor.

**fit**(*self*, *X: pandas.DataFrame*, *y: Optional[pandas.Series] = None*)

Fits estimator to data.

**Parameters**

- **X** (*pd.DataFrame*) – The input training data of shape [n_samples, n_features].
- **y** (*pd.Series, optional*) – The target training data of length [n_samples].

**Returns** self

**get_prediction_intervals**(*self*, *X: pandas.DataFrame*, *y: Optional[pandas.Series] = None*, *coverage: List[float] = None*, *predictions: pandas.Series = None*) → Dict[str, pandas.Series]

Find the prediction intervals using the fitted regressor.

This function takes the predictions of the fitted estimator and calculates the rolling standard deviation across all predictions using a window size of 5. The lower and upper predictions are determined by taking the percent point (quantile) function of the lower tail probability at each bound multiplied by the rolling standard deviation.

**Parameters**

- **X** (*pd.DataFrame*) – Data of shape [n_samples, n_features].
- **y** (*pd.Series*) – Target data. Ignored.
- **coverage** (*list[float]*) – A list of floats between the values 0 and 1 that the upper and lower bounds of the prediction interval should be calculated for.
- **predictions** (*pd.Series*) – Optional list of predictions to use. If None, will generate predictions using *X*.

**Returns** Prediction intervals, keys are in the format {coverage}_lower or {coverage}_upper.

**Return type** dict

**Raises** **MethodPropertyNotFoundError** – If the estimator does not support Time Series Regression as a problem type.

**static load**(*file_path*)

Loads component at file path.

**Parameters** **file_path** (*str*) – Location to load file.

**Returns** ComponentBase object

**needs_fitting**(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need to be fit or whose fit methods do nothing.

**Returns** True.

**property parameters**(*self*)

Returns the parameters which were used to initialize the component.

**predict**(*self*, *X: pandas.DataFrame*) → pandas.Series

Make predictions using selected features.

**Parameters** **X** (*pd.DataFrame*) – Data of shape [n_samples, n_features].

**Returns** Predicted values.

**Return type** pd.Series

**Raises** **MethodPropertyNotFoundError** – If estimator does not have a predict method or a component_obj that implements predict.

**predict_proba**(*self*, *X: pandas.DataFrame*) → pandas.Series

Make probability estimates for labels.

**Parameters** **X** (*pd.DataFrame*) – Features.

**Returns** Probability estimates.

**Return type** pd.Series

**Raises** **MethodPropertyNotFoundError** – If estimator does not have a predict_proba method or a component_obj that implements predict_proba.

**save**(*self*, *file_path*, *pickle_protocol=cloudpickle.DEFAULT_PROTOCOL*)

Saves component at file path.

**Parameters**

- **file_path** (*str*) – Location to save file.
- **pickle_protocol** (*int*) – The pickle data stream format.

**update_parameters**(*self*, *update_dict*, *reset_fit=True*)

Updates the parameter dictionary of the component.

**Parameters**

- **update_dict** (*dict*) – A dict of parameters to update.
- **reset_fit** (*bool, optional*) – If True, will set *_is_fitted* to False.

**class** evalml.pipelines.components.estimators.**Estimator**(*parameters: dict = None*, *component_obj: Type[*evalml.pipelines.components.ComponentBase*] = None*, *random_seed: Union[int, float] = 0*, ***kwargs*)

A component that fits and predicts given data.

To implement a new Estimator, define your own class which is a subclass of Estimator, including a name and a list of acceptable ranges for any parameters to be tuned during the automl search (hyperparameters). Define an *__init__* method which sets up any necessary state and objects. Make sure your *__init__* only uses standard keyword arguments and calls *super().__init__()* with a parameters dict. You may also override the *fit*, *transform*, *fit_transform* and other methods in this class if appropriate.

To see some examples, check out the definitions of any Estimator component subclass.

**Parameters**

- **parameters** (*dict*) – Dictionary of parameters for the component. Defaults to None.
- **component_obj** (*obj*) – Third-party objects useful in component implementation. Defaults to None.
- **random_seed** (*int*) – Seed for the random number generator. Defaults to 0.

**Attributes**

| | |
|---|---|
| **model_family** | ModelFamily.NONE |
| **modifies_features** | True |
| **modifies_target** | False |
| **training_only** | False |

**Methods**

| | |
|---|---|
| *clone* | Constructs a new component with the same parameters and random state. |
| *default_parameters* | Returns the default parameters for this component. |
| *describe* | Describe a component and its parameters. |
| *feature_importance* | Returns importance associated with each feature. |
| *fit* | Fits estimator to data. |
| *get_prediction_intervals* | Find the prediction intervals using the fitted regressor. |
| *load* | Loads component at file path. |
| *model_family* | ModelFamily.NONE |
| *name* | Returns string name of this component. |
| *needs_fitting* | Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances. |
| *parameters* | Returns the parameters which were used to initialize the component. |
| *predict* | Make predictions using selected features. |
| *predict_proba* | Make probability estimates for labels. |
| *save* | Saves component at file path. |
| *supported_problem_types* | Problem types this estimator supports. |
| *update_parameters* | Updates the parameter dictionary of the component. |

**clone**(*self*)

Constructs a new component with the same parameters and random state.

**Returns** A new instance of this component with identical parameters and random state.

**default_parameters**(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

**Returns** Default parameters for this component.

**Return type** dict

**describe**(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

**Parameters**

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description as dictionary in the format {"name": name, "parameters": parameters}

**Returns** Returns dictionary if return_dict is True, else None.

**Return type** None or dict

**property feature_importance**(*self*) → pandas.Series

Returns importance associated with each feature.

**Returns** Importance associated with each feature.

**Return type** np.ndarray

**Raises** **MethodPropertyNotFoundError** – If estimator does not have a feature_importance method or a component_obj that implements feature_importance.

**fit**(*self*, *X: pandas.DataFrame*, *y: Optional[pandas.Series] = None*)

Fits estimator to data.

**Parameters**

- **X** (*pd.DataFrame*) – The input training data of shape [n_samples, n_features].
- **y** (*pd.Series, optional*) – The target training data of length [n_samples].

**Returns** self

**get_prediction_intervals**(*self*, *X: pandas.DataFrame*, *y: Optional[pandas.Series] = None*, *coverage: List[float] = None*, *predictions: pandas.Series = None*) → Dict[str, pandas.Series]

Find the prediction intervals using the fitted regressor.

This function takes the predictions of the fitted estimator and calculates the rolling standard deviation across all predictions using a window size of 5. The lower and upper predictions are determined by taking the percent point (quantile) function of the lower tail probability at each bound multiplied by the rolling standard deviation.

**Parameters**

- **X** (*pd.DataFrame*) – Data of shape [n_samples, n_features].
- **y** (*pd.Series*) – Target data. Ignored.
- **coverage** (*list[float]*) – A list of floats between the values 0 and 1 that the upper and lower bounds of the prediction interval should be calculated for.

- **predictions** (*pd.Series*) – Optional list of predictions to use. If None, will generate predictions using *X*.

**Returns** Prediction intervals, keys are in the format {coverage}_lower or {coverage}_upper.

**Return type** dict

**Raises** **MethodPropertyNotFoundError** – If the estimator does not support Time Series Regression as a problem type.

**static load**(*file_path*)

Loads component at file path.

**Parameters** **file_path** (*str*) – Location to load file.

**Returns** ComponentBase object

**property model_family**(*cls*)

Returns ModelFamily of this component.

**property name**(*cls*)

Returns string name of this component.

**needs_fitting**(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need to be fit or whose fit methods do nothing.

**Returns** True.

**property parameters**(*self*)

Returns the parameters which were used to initialize the component.

**predict**(*self*, *X: pandas.DataFrame*) → pandas.Series

Make predictions using selected features.

**Parameters** **X** (*pd.DataFrame*) – Data of shape [n_samples, n_features].

**Returns** Predicted values.

**Return type** pd.Series

**Raises** **MethodPropertyNotFoundError** – If estimator does not have a predict method or a component_obj that implements predict.

**predict_proba**(*self*, *X: pandas.DataFrame*) → pandas.Series

Make probability estimates for labels.

**Parameters** **X** (*pd.DataFrame*) – Features.

**Returns** Probability estimates.

**Return type** pd.Series

**Raises** **MethodPropertyNotFoundError** – If estimator does not have a predict_proba method or a component_obj that implements predict_proba.

**save**(*self*, *file_path*, *pickle_protocol=cloudpickle.DEFAULT_PROTOCOL*)

Saves component at file path.

**Parameters**

- **file_path** (*str*) – Location to save file.

- **pickle_protocol** (*int*) – The pickle data stream format.

**property supported_problem_types**(*cls*)

Problem types this estimator supports.

**update_parameters**(*self*, *update_dict*, *reset_fit=True*)

Updates the parameter dictionary of the component.

**Parameters**

- **update_dict** (*dict*) – A dict of parameters to update.
- **reset_fit** (*bool, optional*) – If True, will set *_is_fitted* to False.

**class** evalml.pipelines.components.estimators.**ExponentialSmoothingRegressor**(*trend: Optional[str] = None, damped_trend: bool = False, seasonal: Optional[str] = None, sp: int = 2, n_jobs: int = - 1, random_seed: Union[int, float] = 0, **kwargs*)

Holt-Winters Exponential Smoothing Forecaster.

Currently ExponentialSmoothingRegressor isn't supported via conda install. It's recommended that it be installed via PyPI.

**Parameters**

- **trend** (*str*) – Type of trend component. Defaults to None.
- **damped_trend** (*bool*) – If the trend component should be damped. Defaults to False.
- **seasonal** (*str*) – Type of seasonal component. Takes one of {"additive", None}. Can also be multiplicative if
- **0** (*none of the target data is*) –
- **None.** (*but AutoMLSearch wiill not tune for this. Defaults to*) –
- **sp** (*int*) – The number of seasonal periods to consider. Defaults to 2.
- **n_jobs** (*int or None*) – Non-negative integer describing level of parallelism used for pipelines. Defaults to -1.
- **random_seed** (*int*) – Seed for the random number generator. Defaults to 0.

**Attributes**

| | |
|---|---|
| **hyper-parameter_ranges** | { "trend": [None, "additive"], "damped_trend": [True, False], "seasonal": [None, "additive"], "sp": Integer(2, 8),} |
| **model_family** | ModelFamily.EXPONENTIAL_SMOOTHING |
| **modifies_features** | True |
| **modifies_target** | False |
| **name** | Exponential Smoothing Regressor |
| **supported_problem_types** | [ProblemTypes.TIME_SERIES_REGRESSION] |
| **training_only** | False |

**Methods**

| | |
|---|---|
| *clone* | Constructs a new component with the same parameters and random state. |
| *default_parameters* | Returns the default parameters for this component. |
| *describe* | Describe a component and its parameters. |
| *feature_importance* | Returns array of 0's with a length of 1 as feature_importance is not defined for Exponential Smoothing regressor. |
| *fit* | Fits Exponential Smoothing Regressor to data. |
| *get_prediction_intervals* | Find the prediction intervals using the fitted ExponentialSmoothingRegressor. |
| *load* | Loads component at file path. |
| *needs_fitting* | Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances. |
| *parameters* | Returns the parameters which were used to initialize the component. |
| *predict* | Make predictions using fitted Exponential Smoothing regressor. |
| *predict_proba* | Make probability estimates for labels. |
| *save* | Saves component at file path. |
| *update_parameters* | Updates the parameter dictionary of the component. |

**clone**(*self*)

Constructs a new component with the same parameters and random state.

**Returns** A new instance of this component with identical parameters and random state.

**default_parameters**(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

**Returns** Default parameters for this component.

**Return type** dict

**describe**(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

**Parameters**

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description as dictionary in the format {"name": name, "parameters": parameters}

**Returns** Returns dictionary if return_dict is True, else None.

**Return type** None or dict

**property feature_importance**(*self*) → pandas.Series

Returns array of 0's with a length of 1 as feature_importance is not defined for Exponential Smoothing regressor.

**fit**(*self, X: pandas.DataFrame, y: Optional[pandas.Series] = None*)

Fits Exponential Smoothing Regressor to data.

**Parameters**

- **X** (*pd.DataFrame*) – The input training data of shape [n_samples, n_features]. Ignored.
- **y** (*pd.Series*) – The target training data of length [n_samples].

**Returns** self

**Raises** **ValueError** – If y was not passed in.

**get_prediction_intervals**(*self, X: pandas.DataFrame, y: Optional[pandas.Series] = None, coverage: List[float] = None, predictions: pandas.Series = None*) → Dict[str, pandas.Series]

Find the prediction intervals using the fitted ExponentialSmoothingRegressor.

Calculates the prediction intervals by using a simulation of the time series following a specified state space model.

**Parameters**

- **X** (*pd.DataFrame*) – Data of shape [n_samples, n_features].
- **y** (*pd.Series*) – Target data. Optional.
- **coverage** (*List[float]*) – A list of floats between the values 0 and 1 that the upper and lower bounds of the prediction interval should be calculated for.
- **predictions** (*pd.Series*) – Not used for Exponential Smoothing regressor.

**Returns** Prediction intervals, keys are in the format {coverage}_lower or {coverage}_upper.

**Return type** dict

**static load**(*file_path*)

Loads component at file path.

**Parameters** **file_path** (*str*) – Location to load file.

**Returns** ComponentBase object

**needs_fitting**(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need to be fit or whose fit methods do nothing.

**Returns** True.

**property parameters**(*self*)

Returns the parameters which were used to initialize the component.

**predict**(*self*, *X: pandas.DataFrame*, *y: Optional[pandas.Series] = None*) → pandas.Series

Make predictions using fitted Exponential Smoothing regressor.

**Parameters**

- **X** (*pd.DataFrame*) – Data of shape [n_samples, n_features]. Ignored except to set forecast horizon.
- **y** (*pd.Series*) – Target data.

**Returns** Predicted values.

**Return type** pd.Series

**predict_proba**(*self*, *X: pandas.DataFrame*) → pandas.Series

Make probability estimates for labels.

**Parameters** **X** (*pd.DataFrame*) – Features.

**Returns** Probability estimates.

**Return type** pd.Series

**Raises** **MethodPropertyNotFoundError** – If estimator does not have a predict_proba method or a component_obj that implements predict_proba.

**save**(*self*, *file_path*, *pickle_protocol=cloudpickle.DEFAULT_PROTOCOL*)

Saves component at file path.

**Parameters**

- **file_path** (*str*) – Location to save file.
- **pickle_protocol** (*int*) – The pickle data stream format.

**update_parameters**(*self*, *update_dict*, *reset_fit=True*)

Updates the parameter dictionary of the component.

**Parameters**

- **update_dict** (*dict*) – A dict of parameters to update.
- **reset_fit** (*bool, optional*) – If True, will set *_is_fitted* to False.

**class** evalml.pipelines.components.estimators.**ExtraTreesClassifier**(*n_estimators=100*, *max_features='auto'*, *max_depth=6*, *min_samples_split=2*, *min_weight_fraction_leaf=0.0*, *n_jobs=- 1*, *random_seed=0*, ***kwargs*)

Extra Trees Classifier.

**Parameters**

- **n_estimators** (*float*) – The number of trees in the forest. Defaults to 100.
- **max_features** (*int, float or {"auto", "sqrt", "log2"}*) – The number of features to consider when looking for the best split:
  - If int, then consider max_features features at each split.

- If float, then max_features is a fraction and int(max_features * n_features) features are considered at each split.
- If "auto", then max_features=sqrt(n_features).
- If "sqrt", then max_features=sqrt(n_features).
- If "log2", then max_features=log2(n_features).
- If None, then max_features = n_features.

The search for a split does not stop until at least one valid partition of the node samples is found, even if it requires to effectively inspect more than max_features features. Defaults to "auto".

- **max_depth** (*int*) – The maximum depth of the tree. Defaults to 6.
- **min_samples_split** (*int or float*) – The minimum number of samples required to split an internal node:
  - If int, then consider min_samples_split as the minimum number.
  - If float, then min_samples_split is a fraction and ceil(min_samples_split * n_samples) are the minimum number of samples for each split.
- **2.** (*Defaults to*) –
- **min_weight_fraction_leaf** (*float*) – The minimum weighted fraction of the sum total of weights (of all the input samples) required to be at a leaf node. Defaults to 0.0.
- **n_jobs** (*int or None*) – Number of jobs to run in parallel. -1 uses all processes. Defaults to -1.
- **random_seed** (*int*) – Seed for the random number generator. Defaults to 0.

**Attributes**

| | |
|---|---|
| **hyperparameter_ranges** | { "n_estimators": Integer(10, 1000), "max_features": ["auto", "sqrt", "log2"], "max_depth": Integer(4, 10),} |
| **model_family** | ModelFamily.EXTRA_TREES |
| **modifies_features** | True |
| **modifies_target** | False |
| **name** | Extra Trees Classifier |
| **supported_problem_types** | [ ProblemTypes.BINARY, ProblemTypes.MULTICLASS, ProblemTypes.TIME_SERIES_BINARY, ProblemTypes.TIME_SERIES_MULTICLASS,] |
| **training_only** | False |

**Methods**

| | |
|---|---|
| *clone* | Constructs a new component with the same parameters and random state. |
| *default_parameters* | Returns the default parameters for this component. |
| *describe* | Describe a component and its parameters. |
| *feature_importance* | Returns importance associated with each feature. |
| *fit* | Fits estimator to data. |
| *get_prediction_intervals* | Find the prediction intervals using the fitted regressor. |
| *load* | Loads component at file path. |
| *needs_fitting* | Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances. |
| *parameters* | Returns the parameters which were used to initialize the component. |
| *predict* | Make predictions using selected features. |
| *predict_proba* | Make probability estimates for labels. |
| *save* | Saves component at file path. |
| *update_parameters* | Updates the parameter dictionary of the component. |

**clone**(*self*)

Constructs a new component with the same parameters and random state.

**Returns** A new instance of this component with identical parameters and random state.

**default_parameters**(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

**Returns** Default parameters for this component.

**Return type** dict

**describe**(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

**Parameters**

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description as dictionary in the format {"name": name, "parameters": parameters}

**Returns** Returns dictionary if return_dict is True, else None.

**Return type** None or dict

**property feature_importance**(*self*) → pandas.Series

Returns importance associated with each feature.

**Returns** Importance associated with each feature.

**Return type** np.ndarray

**Raises** **MethodPropertyNotFoundError** – If estimator does not have a feature_importance method or a component_obj that implements feature_importance.

**fit**(*self*, *X: pandas.DataFrame*, *y: Optional[pandas.Series] = None*)

Fits estimator to data.

**Parameters**

- **X** (*pd.DataFrame*) – The input training data of shape [n_samples, n_features].
- **y** (*pd.Series, optional*) – The target training data of length [n_samples].

**Returns** self

**get_prediction_intervals**(*self, X: pandas.DataFrame, y: Optional[pandas.Series] = None, coverage: List[float] = None, predictions: pandas.Series = None*) → Dict[str, pandas.Series]

Find the prediction intervals using the fitted regressor.

This function takes the predictions of the fitted estimator and calculates the rolling standard deviation across all predictions using a window size of 5. The lower and upper predictions are determined by taking the percent point (quantile) function of the lower tail probability at each bound multiplied by the rolling standard deviation.

**Parameters**

- **X** (*pd.DataFrame*) – Data of shape [n_samples, n_features].
- **y** (*pd.Series*) – Target data. Ignored.
- **coverage** (*list[float]*) – A list of floats between the values 0 and 1 that the upper and lower bounds of the prediction interval should be calculated for.
- **predictions** (*pd.Series*) – Optional list of predictions to use. If None, will generate predictions using *X*.

**Returns** Prediction intervals, keys are in the format {coverage}_lower or {coverage}_upper.

**Return type** dict

**Raises** **MethodPropertyNotFoundError** – If the estimator does not support Time Series Regression as a problem type.

**static load**(*file_path*)

Loads component at file path.

**Parameters** **file_path** (*str*) – Location to load file.

**Returns** ComponentBase object

**needs_fitting**(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need to be fit or whose fit methods do nothing.

**Returns** True.

**property parameters**(*self*)

Returns the parameters which were used to initialize the component.

**predict**(*self, X: pandas.DataFrame*) → pandas.Series

Make predictions using selected features.

**Parameters** **X** (*pd.DataFrame*) – Data of shape [n_samples, n_features].

**Returns** Predicted values.

**Return type** pd.Series

**Raises** `MethodPropertyNotFoundError` – If estimator does not have a predict method or a component_obj that implements predict.

`predict_proba`(*self*, *X: pandas.DataFrame*) → pandas.Series

Make probability estimates for labels.

**Parameters** `X` (*pd.DataFrame*) – Features.

**Returns** Probability estimates.

**Return type** pd.Series

**Raises** `MethodPropertyNotFoundError` – If estimator does not have a predict_proba method or a component_obj that implements predict_proba.

`save`(*self*, *file_path*, *pickle_protocol=cloudpickle.DEFAULT_PROTOCOL*)

Saves component at file path.

**Parameters**

- `file_path` (*str*) – Location to save file.
- `pickle_protocol` (*int*) – The pickle data stream format.

`update_parameters`(*self*, *update_dict*, *reset_fit=True*)

Updates the parameter dictionary of the component.

**Parameters**

- `update_dict` (*dict*) – A dict of parameters to update.
- `reset_fit` (*bool, optional*) – If True, will set *_is_fitted* to False.

**class** `evalml.pipelines.components.estimators.`**`ExtraTreesRegressor`**(*n_estimators: int = 100*, *max_features: str = 'auto'*, *max_depth: int = 6*, *min_samples_split: int = 2*, *min_weight_fraction_leaf: float = 0.0*, *n_jobs: int = - 1*, *random_seed: Union[int, float] = 0*, ***kwargs*)

Extra Trees Regressor.

**Parameters**

- `n_estimators` (*float*) – The number of trees in the forest. Defaults to 100.
- `max_features` (*int, float or {"auto", "sqrt", "log2"}*) – The number of features to consider when looking for the best split:
  - If int, then consider max_features features at each split.
  - If float, then max_features is a fraction and int(max_features * n_features) features are considered at each split.
  - If "auto", then max_features=sqrt(n_features).
  - If "sqrt", then max_features=sqrt(n_features).
  - If "log2", then max_features=log2(n_features).
  - If None, then max_features = n_features.

The search for a split does not stop until at least one valid partition of the node samples is found, even if it requires to effectively inspect more than max_features features. Defaults to "auto".

- **max_depth** (*int*) – The maximum depth of the tree. Defaults to 6.
- **min_samples_split** (*int or float*) – The minimum number of samples required to split an internal node:
  - If int, then consider min_samples_split as the minimum number.
  - If float, then min_samples_split is a fraction and ceil(min_samples_split * n_samples) are the minimum number of samples for each split.
- **2.** (*Defaults to*) –
- **min_weight_fraction_leaf** (*float*) – The minimum weighted fraction of the sum total of weights (of all the input samples) required to be at a leaf node. Defaults to 0.0.
- **n_jobs** (*int or None*) – Number of jobs to run in parallel. -1 uses all processes. Defaults to -1.
- **random_seed** (*int*) – Seed for the random number generator. Defaults to 0.

**Attributes**

| | |
|---|---|
| **hyper-parameter_ranges** | { "n_estimators": Integer(10, 1000), "max_features": ["auto", "sqrt", "log2"], "max_depth": Integer(4, 10),} |
| **model_family** | ModelFamily.EXTRA_TREES |
| **modifies_features** | True |
| **modifies_target** | False |
| **name** | Extra Trees Regressor |
| **supported_problem_types** | [ ProblemTypes.REGRESSION, ProblemTypes.TIME_SERIES_REGRESSION,] |
| **training_only** | False |

**Methods**

| | |
|---|---|
| *clone* | Constructs a new component with the same parameters and random state. |
| *default_parameters* | Returns the default parameters for this component. |
| *describe* | Describe a component and its parameters. |
| *feature_importance* | Returns importance associated with each feature. |
| *fit* | Fits estimator to data. |
| *get_prediction_intervals* | Find the prediction intervals using the fitted ExtraTreesRegressor. |
| *load* | Loads component at file path. |
| *needs_fitting* | Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances. |
| *parameters* | Returns the parameters which were used to initialize the component. |
| *predict* | Make predictions using selected features. |
| *predict_proba* | Make probability estimates for labels. |
| *save* | Saves component at file path. |
| *update_parameters* | Updates the parameter dictionary of the component. |

**clone**(*self*)

Constructs a new component with the same parameters and random state.

**Returns** A new instance of this component with identical parameters and random state.

**default_parameters**(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

**Returns** Default parameters for this component.

**Return type** dict

**describe**(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

**Parameters**

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description as dictionary in the format {"name": name, "parameters": parameters}

**Returns** Returns dictionary if return_dict is True, else None.

**Return type** None or dict

**property feature_importance**(*self*) → pandas.Series

Returns importance associated with each feature.

**Returns** Importance associated with each feature.

**Return type** np.ndarray

**Raises** **MethodPropertyNotFoundError** – If estimator does not have a feature_importance method or a component_obj that implements feature_importance.

**fit**(*self*, *X: pandas.DataFrame*, *y: Optional[pandas.Series] = None*)

Fits estimator to data.

**Parameters**

- **X** (*pd.DataFrame*) – The input training data of shape [n_samples, n_features].
- **y** (*pd.Series, optional*) – The target training data of length [n_samples].

**Returns** self

**get_prediction_intervals**(*self, X: pandas.DataFrame, y: Optional[pandas.Series] = None, coverage: List[float] = None, predictions: pandas.Series = None*) → Dict[str, pandas.Series]

Find the prediction intervals using the fitted ExtraTreesRegressor.

**Parameters**

- **X** (*pd.DataFrame*) – Data of shape [n_samples, n_features].
- **y** (*pd.Series*) – Target data. Optional.
- **coverage** (*list[float]*) – A list of floats between the values 0 and 1 that the upper and lower bounds of the prediction interval should be calculated for.
- **predictions** (*pd.Series*) – Optional list of predictions to use. If None, will generate predictions using *X*.

**Returns** Prediction intervals, keys are in the format {coverage}_lower or {coverage}_upper.

**Return type** dict

**static load**(*file_path*)

Loads component at file path.

**Parameters** **file_path** (*str*) – Location to load file.

**Returns** ComponentBase object

**needs_fitting**(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need to be fit or whose fit methods do nothing.

**Returns** True.

**property parameters**(*self*)

Returns the parameters which were used to initialize the component.

**predict**(*self, X: pandas.DataFrame*) → pandas.Series

Make predictions using selected features.

**Parameters** **X** (*pd.DataFrame*) – Data of shape [n_samples, n_features].

**Returns** Predicted values.

**Return type** pd.Series

**Raises** **MethodPropertyNotFoundError** – If estimator does not have a predict method or a component_obj that implements predict.

**predict_proba**(*self, X: pandas.DataFrame*) → pandas.Series

Make probability estimates for labels.

**Parameters** **X** (*pd.DataFrame*) – Features.

**Returns** Probability estimates.

**Return type** pd.Series

**Raises** `MethodPropertyNotFoundError` – If estimator does not have a predict_proba method or a component_obj that implements predict_proba.

`save`(*self*, *file_path*, *pickle_protocol=cloudpickle.DEFAULT_PROTOCOL*)

Saves component at file path.

**Parameters**

- `file_path` (*str*) – Location to save file.
- `pickle_protocol` (*int*) – The pickle data stream format.

`update_parameters`(*self*, *update_dict*, *reset_fit=True*)

Updates the parameter dictionary of the component.

**Parameters**

- `update_dict` (*dict*) – A dict of parameters to update.
- `reset_fit` (*bool, optional*) – If True, will set *_is_fitted* to False.

`class evalml.pipelines.components.estimators.KNeighborsClassifier`(*n_neighbors=5*, *weights='uniform'*, *algorithm='auto'*, *leaf_size=30*, *p=2*, *random_seed=0*, ***kwargs*)

K-Nearest Neighbors Classifier.

**Parameters**

- `n_neighbors` (*int*) – Number of neighbors to use by default. Defaults to 5.
- `weights` (*{'uniform', 'distance'} or callable*) – Weight function used in prediction. Can be:
  - 'uniform' : uniform weights. All points in each neighborhood are weighted equally.
  - 'distance' : weight points by the inverse of their distance. in this case, closer neighbors of a query point will have a greater influence than neighbors which are further away.
  - [callable] : a user-defined function which accepts an array of distances, and returns an array of the same shape containing the weights.

  Defaults to "uniform".
- `algorithm` (*{'auto', 'ball_tree', 'kd_tree', 'brute'}*) – Algorithm used to compute the nearest neighbors:
  - 'ball_tree' will use BallTree
  - 'kd_tree' will use KDTree
  - 'brute' will use a brute-force search.

  'auto' will attempt to decide the most appropriate algorithm based on the values passed to fit method. Defaults to "auto". Note: fitting on sparse input will override the setting of this parameter, using brute force.
- `leaf_size` (*int*) – Leaf size passed to BallTree or KDTree. This can affect the speed of the construction and query, as well as the memory required to store the tree. The optimal value depends on the nature of the problem. Defaults to 30.

- **p** (*int*) – Power parameter for the Minkowski metric. When p = 1, this is equivalent to using manhattan_distance (l1), and euclidean_distance (l2) for p = 2. For arbitrary p, minkowski_distance (l_p) is used. Defaults to 2.
- **random_seed** (*int*) – Seed for the random number generator. Defaults to 0.

**Attributes**

| | |
|---|---|
| **hyper-parameter_ranges** | { "n_neighbors": Integer(2, 12), "weights": ["uniform", "distance"], "algorithm": ["auto", "ball_tree", "kd_tree", "brute"], "leaf_size": Integer(10, 30), "p": Integer(1, 5),} |
| **model_family** | ModelFamily.K_NEIGHBORS |
| **modifies_features** | True |
| **modifies_target** | False |
| **name** | KNN Classifier |
| **supported_problem_types** | [ ProblemTypes.BINARY, ProblemTypes.MULTICLASS, ProblemTypes.TIME_SERIES_BINARY, ProblemTypes.TIME_SERIES_MULTICLASS,] |
| **training_only** | False |

**Methods**

| | |
|---|---|
| *clone* | Constructs a new component with the same parameters and random state. |
| *default_parameters* | Returns the default parameters for this component. |
| *describe* | Describe a component and its parameters. |
| *feature_importance* | Returns array of 0's matching the input number of features as feature_importance is not defined for KNN classifiers. |
| *fit* | Fits estimator to data. |
| *get_prediction_intervals* | Find the prediction intervals using the fitted regressor. |
| *load* | Loads component at file path. |
| *needs_fitting* | Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances. |
| *parameters* | Returns the parameters which were used to initialize the component. |
| *predict* | Make predictions using selected features. |
| *predict_proba* | Make probability estimates for labels. |
| *save* | Saves component at file path. |
| *update_parameters* | Updates the parameter dictionary of the component. |

**clone**(*self*)

Constructs a new component with the same parameters and random state.

**Returns** A new instance of this component with identical parameters and random state.

**default_parameters**(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

**Returns** Default parameters for this component.

**Return type** dict

`describe`(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

**Parameters**

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description as dictionary in the format {"name": name, "parameters": parameters}

**Returns** Returns dictionary if return_dict is True, else None.

**Return type** None or dict

`property feature_importance`(*self*)

Returns array of 0's matching the input number of features as feature_importance is not defined for KNN classifiers.

`fit`(*self*, *X: pandas.DataFrame*, *y: Optional[pandas.Series] = None*)

Fits estimator to data.

**Parameters**

- **X** (*pd.DataFrame*) – The input training data of shape [n_samples, n_features].
- **y** (*pd.Series, optional*) – The target training data of length [n_samples].

**Returns** self

`get_prediction_intervals`(*self*, *X: pandas.DataFrame*, *y: Optional[pandas.Series] = None*, *coverage: List[float] = None*, *predictions: pandas.Series = None*) → Dict[str, pandas.Series]

Find the prediction intervals using the fitted regressor.

This function takes the predictions of the fitted estimator and calculates the rolling standard deviation across all predictions using a window size of 5. The lower and upper predictions are determined by taking the percent point (quantile) function of the lower tail probability at each bound multiplied by the rolling standard deviation.

**Parameters**

- **X** (*pd.DataFrame*) – Data of shape [n_samples, n_features].
- **y** (*pd.Series*) – Target data. Ignored.
- **coverage** (*list[float]*) – A list of floats between the values 0 and 1 that the upper and lower bounds of the prediction interval should be calculated for.
- **predictions** (*pd.Series*) – Optional list of predictions to use. If None, will generate predictions using *X*.

**Returns** Prediction intervals, keys are in the format {coverage}_lower or {coverage}_upper.

**Return type** dict

**Raises** **MethodPropertyNotFoundError** – If the estimator does not support Time Series Regression as a problem type.

**static load**(*file_path*)

Loads component at file path.

**Parameters** **file_path** (*str*) – Location to load file.

**Returns** ComponentBase object

**needs_fitting**(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need to be fit or whose fit methods do nothing.

**Returns** True.

**property parameters**(*self*)

Returns the parameters which were used to initialize the component.

**predict**(*self*, *X: pandas.DataFrame*) → pandas.Series

Make predictions using selected features.

**Parameters** **X** (*pd.DataFrame*) – Data of shape [n_samples, n_features].

**Returns** Predicted values.

**Return type** pd.Series

**Raises** **MethodPropertyNotFoundError** – If estimator does not have a predict method or a component_obj that implements predict.

**predict_proba**(*self*, *X: pandas.DataFrame*) → pandas.Series

Make probability estimates for labels.

**Parameters** **X** (*pd.DataFrame*) – Features.

**Returns** Probability estimates.

**Return type** pd.Series

**Raises** **MethodPropertyNotFoundError** – If estimator does not have a predict_proba method or a component_obj that implements predict_proba.

**save**(*self*, *file_path*, *pickle_protocol=cloudpickle.DEFAULT_PROTOCOL*)

Saves component at file path.

**Parameters**

- **file_path** (*str*) – Location to save file.
- **pickle_protocol** (*int*) – The pickle data stream format.

**update_parameters**(*self*, *update_dict*, *reset_fit=True*)

Updates the parameter dictionary of the component.

**Parameters**

- **update_dict** (*dict*) – A dict of parameters to update.
- **reset_fit** (*bool, optional*) – If True, will set *_is_fitted* to False.

**class** evalml.pipelines.components.estimators.**LightGBMClassifier**(*boosting_type='gbdt', learning_rate=0.1, n_estimators=100, max_depth=0, num_leaves=31, min_child_samples=20, bagging_fraction=0.9, bagging_freq=0, n_jobs=- 1, random_seed=0, **kwargs*)

LightGBM Classifier.

**Parameters**

- **boosting_type** (*string*) – Type of boosting to use. Defaults to "gbdt". - 'gbdt' uses traditional Gradient Boosting Decision Tree - "dart", uses Dropouts meet Multiple Additive Regression Trees - "goss", uses Gradient-based One-Side Sampling - "rf", uses Random Forest
- **learning_rate** (*float*) – Boosting learning rate. Defaults to 0.1.
- **n_estimators** (*int*) – Number of boosted trees to fit. Defaults to 100.
- **max_depth** (*int*) – Maximum tree depth for base learners, <=0 means no limit. Defaults to 0.
- **num_leaves** (*int*) – Maximum tree leaves for base learners. Defaults to 31.
- **min_child_samples** (*int*) – Minimum number of data needed in a child (leaf). Defaults to 20.
- **bagging_fraction** (*float*) – LightGBM will randomly select a subset of features on each iteration (tree) without resampling if this is smaller than 1.0. For example, if set to 0.8, LightGBM will select 80% of features before training each tree. This can be used to speed up training and deal with overfitting. Defaults to 0.9.
- **bagging_freq** (*int*) – Frequency for bagging. 0 means bagging is disabled. k means perform bagging at every k iteration. Every k-th iteration, LightGBM will randomly select bagging_fraction * 100 % of the data to use for the next k iterations. Defaults to 0.
- **n_jobs** (*int or None*) – Number of threads to run in parallel. -1 uses all threads. Defaults to -1.
- **random_seed** (*int*) – Seed for the random number generator. Defaults to 0.

**Attributes**

| | |
|---|---|
| **hyper-parameter_ranges** | { "learning_rate": Real(0.000001, 1), "boosting_type": ["gbdt", "dart", "goss", "rf"], "n_estimators": Integer(10, 100), "max_depth": Integer(0, 10), "num_leaves": Integer(2, 100), "min_child_samples": Integer(1, 100), "bagging_fraction": Real(0.000001, 1), "bagging_freq": Integer(0, 1),} |
| **model_family** | ModelFamily.LIGHTGBM |
| **modifies_features** | True |
| **modifies_target** | False |
| **name** | LightGBM Classifier |
| **SEED_MAX** | SEED_BOUNDS.max_bound |
| **SEED_MIN** | 0 |
| **supported_problem_types** | [ ProblemTypes.BINARY, ProblemTypes.MULTICLASS, ProblemTypes.TIME_SERIES_BINARY, ProblemTypes.TIME_SERIES_MULTICLASS,] |
| **training_only** | False |

**Methods**

| | |
|---|---|
| *clone* | Constructs a new component with the same parameters and random state. |
| *default_parameters* | Returns the default parameters for this component. |
| *describe* | Describe a component and its parameters. |
| *feature_importance* | Returns importance associated with each feature. |
| *fit* | Fits LightGBM classifier component to data. |
| *get_prediction_intervals* | Find the prediction intervals using the fitted regressor. |
| *load* | Loads component at file path. |
| *needs_fitting* | Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances. |
| *parameters* | Returns the parameters which were used to initialize the component. |
| *predict* | Make predictions using the fitted LightGBM classifier. |
| *predict_proba* | Make prediction probabilities using the fitted LightGBM classifier. |
| *save* | Saves component at file path. |
| *update_parameters* | Updates the parameter dictionary of the component. |

**clone**(*self*)

Constructs a new component with the same parameters and random state.

**Returns** A new instance of this component with identical parameters and random state.

**default_parameters**(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

**Returns** Default parameters for this component.

**Return type** dict

**describe**(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

**Parameters**

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description as dictionary in the format {"name": name, "parameters": parameters}

**Returns** Returns dictionary if return_dict is True, else None.

**Return type** None or dict

**property feature_importance**(*self*) → pandas.Series

Returns importance associated with each feature.

**Returns** Importance associated with each feature.

**Return type** np.ndarray

**Raises** **MethodPropertyNotFoundError** – If estimator does not have a feature_importance method or a component_obj that implements feature_importance.

**fit**(*self*, *X*, *y=None*)

Fits LightGBM classifier component to data.

**Parameters**

- **X** (*pd.DataFrame*) – The input training data of shape [n_samples, n_features].
- **y** (*pd.Series*) – The target training data of length [n_samples].

**Returns** self

**get_prediction_intervals**(*self*, *X: pandas.DataFrame*, *y: Optional[pandas.Series] = None*, *coverage: List[float] = None*, *predictions: pandas.Series = None*) → Dict[str, pandas.Series]

Find the prediction intervals using the fitted regressor.

This function takes the predictions of the fitted estimator and calculates the rolling standard deviation across all predictions using a window size of 5. The lower and upper predictions are determined by taking the percent point (quantile) function of the lower tail probability at each bound multiplied by the rolling standard deviation.

**Parameters**

- **X** (*pd.DataFrame*) – Data of shape [n_samples, n_features].
- **y** (*pd.Series*) – Target data. Ignored.
- **coverage** (*list[float]*) – A list of floats between the values 0 and 1 that the upper and lower bounds of the prediction interval should be calculated for.
- **predictions** (*pd.Series*) – Optional list of predictions to use. If None, will generate predictions using *X*.

**Returns** Prediction intervals, keys are in the format {coverage}_lower or {coverage}_upper.

**Return type** dict

**Raises** **MethodPropertyNotFoundError** – If the estimator does not support Time Series Regression as a problem type.

**static load**(*file_path*)

Loads component at file path.

**Parameters** **file_path** (*str*) – Location to load file.

**Returns** ComponentBase object

**needs_fitting**(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need to be fit or whose fit methods do nothing.

**Returns** True.

**property parameters**(*self*)

Returns the parameters which were used to initialize the component.

**predict**(*self*, *X*)

Make predictions using the fitted LightGBM classifier.

**Parameters** **X** (*pd.DataFrame*) – Data of shape [n_samples, n_features].

**Returns** Predicted values.

**Return type** pd.DataFrame

**predict_proba**(*self*, *X*)

Make prediction probabilities using the fitted LightGBM classifier.

**Parameters** **X** (*pd.DataFrame*) – Data of shape [n_samples, n_features].

**Returns** Predicted probability values.

**Return type** pd.DataFrame

**save**(*self*, *file_path*, *pickle_protocol=cloudpickle.DEFAULT_PROTOCOL*)

Saves component at file path.

**Parameters**

- **file_path** (*str*) – Location to save file.
- **pickle_protocol** (*int*) – The pickle data stream format.

**update_parameters**(*self*, *update_dict*, *reset_fit=True*)

Updates the parameter dictionary of the component.

**Parameters**

- **update_dict** (*dict*) – A dict of parameters to update.
- **reset_fit** (*bool, optional*) – If True, will set *_is_fitted* to False.

**class** evalml.pipelines.components.estimators.**LightGBMRegressor**(*boosting_type='gbdt', learning_rate=0.1, n_estimators=20, max_depth=0, num_leaves=31, min_child_samples=20, bagging_fraction=0.9, bagging_freq=0, n_jobs=- 1, random_seed=0, **kwargs*)

LightGBM Regressor.

**Parameters**

- **boosting_type** (*string*) – Type of boosting to use. Defaults to "gbdt". - 'gbdt' uses traditional Gradient Boosting Decision Tree - "dart", uses Dropouts meet Multiple Additive Regression Trees - "goss", uses Gradient-based One-Side Sampling - "rf", uses Random Forest
- **learning_rate** (*float*) – Boosting learning rate. Defaults to 0.1.
- **n_estimators** (*int*) – Number of boosted trees to fit. Defaults to 100.
- **max_depth** (*int*) – Maximum tree depth for base learners, <=0 means no limit. Defaults to 0.
- **num_leaves** (*int*) – Maximum tree leaves for base learners. Defaults to 31.
- **min_child_samples** (*int*) – Minimum number of data needed in a child (leaf). Defaults to 20.
- **bagging_fraction** (*float*) – LightGBM will randomly select a subset of features on each iteration (tree) without resampling if this is smaller than 1.0. For example, if set to 0.8, LightGBM will select 80% of features before training each tree. This can be used to speed up training and deal with overfitting. Defaults to 0.9.
- **bagging_freq** (*int*) – Frequency for bagging. 0 means bagging is disabled. k means perform bagging at every k iteration. Every k-th iteration, LightGBM will randomly select bagging_fraction * 100 % of the data to use for the next k iterations. Defaults to 0.
- **n_jobs** (*int or None*) – Number of threads to run in parallel. -1 uses all threads. Defaults to -1.
- **random_seed** (*int*) – Seed for the random number generator. Defaults to 0.

**Attributes**

| | |
|---|---|
| **hyper-parameter_ranges** | { "learning_rate": Real(0.000001, 1), "boosting_type": ["gbdt", "dart", "goss", "rf"], "n_estimators": Integer(10, 100), "max_depth": Integer(0, 10), "num_leaves": Integer(2, 100), "min_child_samples": Integer(1, 100), "bagging_fraction": Real(0.000001, 1), "bagging_freq": Integer(0, 1),} |
| **model_family** | ModelFamily.LIGHTGBM |
| **modifies_features** | True |
| **modifies_target** | False |
| **name** | LightGBM Regressor |
| **SEED_MAX** | SEED_BOUNDS.max_bound |
| **SEED_MIN** | 0 |
| **supported_problem_types** | [ProblemTypes.REGRESSION] |
| **training_only** | False |

**Methods**

| | |
|---|---|
| *clone* | Constructs a new component with the same parameters and random state. |
| *default_parameters* | Returns the default parameters for this component. |
| *describe* | Describe a component and its parameters. |
| *feature_importance* | Returns importance associated with each feature. |
| *fit* | Fits LightGBM regressor to data. |
| *get_prediction_intervals* | Find the prediction intervals using the fitted regressor. |
| *load* | Loads component at file path. |
| *needs_fitting* | Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances. |
| *parameters* | Returns the parameters which were used to initialize the component. |
| *predict* | Make predictions using fitted LightGBM regressor. |
| *predict_proba* | Make probability estimates for labels. |
| *save* | Saves component at file path. |
| *update_parameters* | Updates the parameter dictionary of the component. |

**clone**(*self*)

Constructs a new component with the same parameters and random state.

**Returns** A new instance of this component with identical parameters and random state.

**default_parameters**(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

**Returns** Default parameters for this component.

**Return type** dict

**describe**(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

**Parameters**

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description as dictionary in the format {"name": name, "parameters": parameters}

**Returns** Returns dictionary if return_dict is True, else None.

**Return type** None or dict

**property feature_importance**(*self*) → pandas.Series

Returns importance associated with each feature.

**Returns** Importance associated with each feature.

**Return type** np.ndarray

**Raises** **MethodPropertyNotFoundError** – If estimator does not have a feature_importance method or a component_obj that implements feature_importance.

**fit**(*self*, *X*, *y=None*)

Fits LightGBM regressor to data.

**Parameters**

- **X** (*pd.DataFrame*) – The input training data of shape [n_samples, n_features].
- **y** (*pd.Series*) – The target training data of length [n_samples].

**Returns** self

**get_prediction_intervals**(*self, X: pandas.DataFrame, y: Optional[pandas.Series] = None, coverage: List[float] = None, predictions: pandas.Series = None*) → Dict[str, pandas.Series]

Find the prediction intervals using the fitted regressor.

This function takes the predictions of the fitted estimator and calculates the rolling standard deviation across all predictions using a window size of 5. The lower and upper predictions are determined by taking the percent point (quantile) function of the lower tail probability at each bound multiplied by the rolling standard deviation.

**Parameters**

- **X** (*pd.DataFrame*) – Data of shape [n_samples, n_features].
- **y** (*pd.Series*) – Target data. Ignored.
- **coverage** (*list[float]*) – A list of floats between the values 0 and 1 that the upper and lower bounds of the prediction interval should be calculated for.
- **predictions** (*pd.Series*) – Optional list of predictions to use. If None, will generate predictions using *X*.

**Returns** Prediction intervals, keys are in the format {coverage}_lower or {coverage}_upper.

**Return type** dict

**Raises** **MethodPropertyNotFoundError** – If the estimator does not support Time Series Regression as a problem type.

**static load**(*file_path*)

Loads component at file path.

**Parameters** **file_path** (*str*) – Location to load file.

**Returns** ComponentBase object

**needs_fitting**(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need to be fit or whose fit methods do nothing.

**Returns** True.

**property parameters**(*self*)

Returns the parameters which were used to initialize the component.

**predict**(*self, X*)

Make predictions using fitted LightGBM regressor.

**Parameters** **X** (*pd.DataFrame*) – Data of shape [n_samples, n_features].

**Returns** Predicted values.

**Return type** pd.Series

**predict_proba**(*self*, *X: pandas.DataFrame*) → pandas.Series

Make probability estimates for labels.

**Parameters** **X** (*pd.DataFrame*) – Features.

**Returns** Probability estimates.

**Return type** pd.Series

**Raises** **MethodPropertyNotFoundError** – If estimator does not have a predict_proba method or a component_obj that implements predict_proba.

**save**(*self*, *file_path*, *pickle_protocol=cloudpickle.DEFAULT_PROTOCOL*)

Saves component at file path.

**Parameters**

- **file_path** (*str*) – Location to save file.
- **pickle_protocol** (*int*) – The pickle data stream format.

**update_parameters**(*self*, *update_dict*, *reset_fit=True*)

Updates the parameter dictionary of the component.

**Parameters**

- **update_dict** (*dict*) – A dict of parameters to update.
- **reset_fit** (*bool, optional*) – If True, will set *_is_fitted* to False.

**class** evalml.pipelines.components.estimators.**LinearRegressor**(*fit_intercept=True*, *n_jobs=- 1*, *random_seed=0*, ***kwargs*)

Linear Regressor.

**Parameters**

- **fit_intercept** (*boolean*) – Whether to calculate the intercept for this model. If set to False, no intercept will be used in calculations (i.e. data is expected to be centered). Defaults to True.
- **n_jobs** (*int or None*) – Number of jobs to run in parallel. -1 uses all threads. Defaults to -1.
- **random_seed** (*int*) – Seed for the random number generator. Defaults to 0.

**Attributes**

| | |
|---|---|
| **hyper-parameter_ranges** | { "fit_intercept": [True, False],} |
| **model_family** | ModelFamily.LINEAR_MODEL |
| **modifies_features** | True |
| **modifies_target** | False |
| **name** | Linear Regressor |
| **supported_problem_types** | [ ProblemTypes.REGRESSION, ProblemTypes.TIME_SERIES_REGRESSION,] |
| **training_only** | False |

**Methods**

| | |
|---|---|
| *clone* | Constructs a new component with the same parameters and random state. |
| *default_parameters* | Returns the default parameters for this component. |
| *describe* | Describe a component and its parameters. |
| *feature_importance* | Feature importance for fitted linear regressor. |
| *fit* | Fits estimator to data. |
| *get_prediction_intervals* | Find the prediction intervals using the fitted regressor. |
| *load* | Loads component at file path. |
| *needs_fitting* | Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances. |
| *parameters* | Returns the parameters which were used to initialize the component. |
| *predict* | Make predictions using selected features. |
| *predict_proba* | Make probability estimates for labels. |
| *save* | Saves component at file path. |
| *update_parameters* | Updates the parameter dictionary of the component. |

**clone**(*self*)

Constructs a new component with the same parameters and random state.

**Returns** A new instance of this component with identical parameters and random state.

**default_parameters**(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

**Returns** Default parameters for this component.

**Return type** dict

**describe**(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

**Parameters**

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description as dictionary in the format {"name": name, "parameters": parameters}

**Returns** Returns dictionary if return_dict is True, else None.

**Return type** None or dict

**property feature_importance**(*self*)

Feature importance for fitted linear regressor.

**fit**(*self*, *X: pandas.DataFrame*, *y: Optional[pandas.Series] = None*)

Fits estimator to data.

**Parameters**

- **X** (*pd.DataFrame*) – The input training data of shape [n_samples, n_features].
- **y** (*pd.Series, optional*) – The target training data of length [n_samples].

**Returns** self

**get_prediction_intervals**(*self*, *X: pandas.DataFrame*, *y: Optional[pandas.Series] = None*, *coverage: List[float] = None*, *predictions: pandas.Series = None*) → Dict[str, pandas.Series]

Find the prediction intervals using the fitted regressor.

This function takes the predictions of the fitted estimator and calculates the rolling standard deviation across all predictions using a window size of 5. The lower and upper predictions are determined by taking the percent point (quantile) function of the lower tail probability at each bound multiplied by the rolling standard deviation.

**Parameters**

- **X** (*pd.DataFrame*) – Data of shape [n_samples, n_features].
- **y** (*pd.Series*) – Target data. Ignored.
- **coverage** (*list[float]*) – A list of floats between the values 0 and 1 that the upper and lower bounds of the prediction interval should be calculated for.
- **predictions** (*pd.Series*) – Optional list of predictions to use. If None, will generate predictions using *X*.

**Returns** Prediction intervals, keys are in the format {coverage}_lower or {coverage}_upper.

**Return type** dict

**Raises** **MethodPropertyNotFoundError** – If the estimator does not support Time Series Regression as a problem type.

**static load**(*file_path*)

Loads component at file path.

**Parameters** **file_path** (*str*) – Location to load file.

**Returns** ComponentBase object

**needs_fitting**(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need to be fit or whose fit methods do nothing.

**Returns** True.

**property parameters**(*self*)

Returns the parameters which were used to initialize the component.

**predict**(*self*, *X: pandas.DataFrame*) → pandas.Series

Make predictions using selected features.

**Parameters** **X** (*pd.DataFrame*) – Data of shape [n_samples, n_features].

**Returns** Predicted values.

**Return type** pd.Series

**Raises** **MethodPropertyNotFoundError** – If estimator does not have a predict method or a component_obj that implements predict.

**predict_proba**(*self*, *X: pandas.DataFrame*) → pandas.Series

Make probability estimates for labels.

**Parameters** **X** (*pd.DataFrame*) – Features.

**Returns** Probability estimates.

**Return type** pd.Series

**Raises** **MethodPropertyNotFoundError** – If estimator does not have a predict_proba method or a component_obj that implements predict_proba.

**save**(*self*, *file_path*, *pickle_protocol=cloudpickle.DEFAULT_PROTOCOL*)

Saves component at file path.

**Parameters**

- **file_path** (*str*) – Location to save file.
- **pickle_protocol** (*int*) – The pickle data stream format.

**update_parameters**(*self*, *update_dict*, *reset_fit=True*)

Updates the parameter dictionary of the component.

**Parameters**

- **update_dict** (*dict*) – A dict of parameters to update.
- **reset_fit** (*bool, optional*) – If True, will set *_is_fitted* to False.

**class** evalml.pipelines.components.estimators.**LogisticRegressionClassifier**(*penalty='l2'*, *C=1.0*, *multi_class='auto'*, *solver='lbfgs'*, *n_jobs=- 1*, *random_seed=0*, ***kwargs*)

Logistic Regression Classifier.

**Parameters**

- **penalty** (*{"l1", "l2", "elasticnet", "none"}*) – The norm used in penalization. Defaults to "l2".
- **C** (*float*) – Inverse of regularization strength. Must be a positive float. Defaults to 1.0.
- **multi_class** (*{"auto", "ovr", "multinomial"}*) – If the option chosen is "ovr", then a binary problem is fit for each label. For "multinomial" the loss minimised is the multinomial loss fit across the entire probability distribution, even when the data is binary. "multinomial" is unavailable when solver="liblinear". "auto" selects "ovr" if the data is binary, or if solver="liblinear", and otherwise selects "multinomial". Defaults to "auto".
- **solver** (*{"newton-cg", "lbfgs", "liblinear", "sag", "saga"}*) – Algorithm to use in the optimization problem. For small datasets, "liblinear" is a good choice, whereas "sag" and "saga" are faster for large ones. For multiclass problems, only "newton-cg", "sag", "saga" and "lbfgs" handle multinomial loss; "liblinear" is limited to one-versus-rest schemes.
  - "newton-cg", "lbfgs", "sag" and "saga" handle L2 or no penalty
  - "liblinear" and "saga" also handle L1 penalty
  - "saga" also supports "elasticnet" penalty
  - "liblinear" does not support setting penalty='none'

Defaults to "lbfgs".

- **n_jobs** (*int*) – Number of parallel threads used to run xgboost. Note that creating thread contention will significantly slow down the algorithm. Defaults to -1.
- **random_seed** (*int*) – Seed for the random number generator. Defaults to 0.

**Attributes**

| | |
|---|---|
| **hyper-parameter_ranges** | { "penalty": ["l2"], "C": Real(0.01, 10),} |
| **model_family** | ModelFamily.LINEAR_MODEL |
| **modifies_features** | True |
| **modifies_target** | False |
| **name** | Logistic Regression Classifier |
| **supported_problem_types** | [ ProblemTypes.BINARY, ProblemTypes.MULTICLASS, ProblemTypes.TIME_SERIES_BINARY, ProblemTypes.TIME_SERIES_MULTICLASS,] |
| **training_only** | False |

**Methods**

| | |
|---|---|
| *clone* | Constructs a new component with the same parameters and random state. |
| *default_parameters* | Returns the default parameters for this component. |
| *describe* | Describe a component and its parameters. |
| *feature_importance* | Feature importance for fitted logistic regression classifier. |
| *fit* | Fits estimator to data. |
| *get_prediction_intervals* | Find the prediction intervals using the fitted regressor. |
| *load* | Loads component at file path. |
| *needs_fitting* | Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances. |
| *parameters* | Returns the parameters which were used to initialize the component. |
| *predict* | Make predictions using selected features. |
| *predict_proba* | Make probability estimates for labels. |
| *save* | Saves component at file path. |
| *update_parameters* | Updates the parameter dictionary of the component. |

**clone**(*self*)

Constructs a new component with the same parameters and random state.

**Returns** A new instance of this component with identical parameters and random state.

**default_parameters**(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

**Returns** Default parameters for this component.

**Return type** dict

`describe`(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

**Parameters**

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description as dictionary in the format {"name": name, "parameters": parameters}

**Returns** Returns dictionary if return_dict is True, else None.

**Return type** None or dict

`property feature_importance`(*self*)

Feature importance for fitted logistic regression classifier.

`fit`(*self*, *X: pandas.DataFrame*, *y: Optional[pandas.Series] = None*)

Fits estimator to data.

**Parameters**

- **X** (*pd.DataFrame*) – The input training data of shape [n_samples, n_features].
- **y** (*pd.Series, optional*) – The target training data of length [n_samples].

**Returns** self

`get_prediction_intervals`(*self*, *X: pandas.DataFrame*, *y: Optional[pandas.Series] = None*, *coverage: List[float] = None*, *predictions: pandas.Series = None*) → Dict[str, pandas.Series]

Find the prediction intervals using the fitted regressor.

This function takes the predictions of the fitted estimator and calculates the rolling standard deviation across all predictions using a window size of 5. The lower and upper predictions are determined by taking the percent point (quantile) function of the lower tail probability at each bound multiplied by the rolling standard deviation.

**Parameters**

- **X** (*pd.DataFrame*) – Data of shape [n_samples, n_features].
- **y** (*pd.Series*) – Target data. Ignored.
- **coverage** (*list[float]*) – A list of floats between the values 0 and 1 that the upper and lower bounds of the prediction interval should be calculated for.
- **predictions** (*pd.Series*) – Optional list of predictions to use. If None, will generate predictions using *X*.

**Returns** Prediction intervals, keys are in the format {coverage}_lower or {coverage}_upper.

**Return type** dict

**Raises** **MethodPropertyNotFoundError** – If the estimator does not support Time Series Regression as a problem type.

`static load`(*file_path*)

Loads component at file path.

**Parameters** **file_path** (*str*) – Location to load file.

**Returns** ComponentBase object

**needs_fitting**(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need to be fit or whose fit methods do nothing.

**Returns** True.

**property parameters**(*self*)

Returns the parameters which were used to initialize the component.

**predict**(*self*, *X: pandas.DataFrame*) → pandas.Series

Make predictions using selected features.

**Parameters** **X** (*pd.DataFrame*) – Data of shape [n_samples, n_features].

**Returns** Predicted values.

**Return type** pd.Series

**Raises** **MethodPropertyNotFoundError** – If estimator does not have a predict method or a component_obj that implements predict.

**predict_proba**(*self*, *X: pandas.DataFrame*) → pandas.Series

Make probability estimates for labels.

**Parameters** **X** (*pd.DataFrame*) – Features.

**Returns** Probability estimates.

**Return type** pd.Series

**Raises** **MethodPropertyNotFoundError** – If estimator does not have a predict_proba method or a component_obj that implements predict_proba.

**save**(*self*, *file_path*, *pickle_protocol=cloudpickle.DEFAULT_PROTOCOL*)

Saves component at file path.

**Parameters**

- **file_path** (*str*) – Location to save file.
- **pickle_protocol** (*int*) – The pickle data stream format.

**update_parameters**(*self*, *update_dict*, *reset_fit=True*)

Updates the parameter dictionary of the component.

**Parameters**

- **update_dict** (*dict*) – A dict of parameters to update.
- **reset_fit** (*bool, optional*) – If True, will set *_is_fitted* to False.

**class** evalml.pipelines.components.estimators.**ProphetRegressor**(*time_index: Optional[Hashable] = None*, *changepoint_prior_scale: float = 0.05*, *seasonality_prior_scale: int = 10*, *holidays_prior_scale: int = 10*, *seasonality_mode: str = 'additive'*, *stan_backend: str = 'CMDSTANPY'*, *interval_width: float = 0.95*, *random_seed: Union[int, float] = 0*, ***kwargs*)

Prophet is a procedure for forecasting time series data based on an additive model where non-linear trends are fit with yearly, weekly, and daily seasonality, plus holiday effects. It works best with time series that have strong seasonal effects and several seasons of historical data. Prophet is robust to missing data and shifts in the trend, and typically handles outliers well.

More information here: https://facebook.github.io/prophet/

**Parameters**

- **time_index** (*str*) – Specifies the name of the column in X that provides the datetime objects. Defaults to None.
- **changepoint_prior_scale** (*float*) – Determines the strength of the sparse prior for fitting on rate changes. Increasing this value increases the flexibility of the trend. Defaults to 0.05.
- **seasonality_prior_scale** (*int*) – Similar to changepoint_prior_scale. Adjusts the extent to which the seasonality model will fit the data. Defaults to 10.
- **holidays_prior_scale** (*int*) – Similar to changepoint_prior_scale. Adjusts the extent to which holidays will fit the data. Defaults to 10.
- **seasonality_mode** (*str*) – Determines how this component fits the seasonality. Options are "additive" and "multiplicative". Defaults to "additive".
- **stan_backend** (*str*) – Determines the backend that should be used to run Prophet. Options are "CMDSTANPY" and "PYSTAN". Defaults to "CMDSTANPY".
- **interval_width** (*float*) – Determines the confidence of the prediction interval range when calling *get_prediction_intervals*. Accepts values in the range (0,1). Defaults to 0.95.
- **random_seed** (*int*) – Seed for the random number generator. Defaults to 0.

**Attributes**

| | |
|---|---|
| **hyper-parameter_ranges** | { "changepoint_prior_scale": Real(0.001, 0.5), "seasonality_prior_scale": Real(0.01, 10), "holidays_prior_scale": Real(0.01, 10), "seasonality_mode": ["additive", "multiplicative"],} |
| **model_family** | ModelFamily.PROPHET |
| **modifies_features** | True |
| **modifies_target** | False |
| **name** | Prophet Regressor |
| **supported_problem_types** | [ProblemTypes.TIME_SERIES_REGRESSION] |
| **training_only** | False |

**Methods**

| | |
|---|---|
| *build_prophet_df* | Build the Prophet data to pass fit and predict on. |
| *clone* | Constructs a new component with the same parameters and random state. |
| *default_parameters* | Returns the default parameters for this component. |
| *describe* | Describe a component and its parameters. |
| *feature_importance* | Returns array of 0's with len(1) as feature_importance is not defined for Prophet regressor. |
| *fit* | Fits Prophet regressor component to data. |
| *get_params* | Get parameters for the Prophet regressor. |
| *get_prediction_intervals* | Find the prediction intervals using the fitted ProphetRegressor. |
| *load* | Loads component at file path. |
| *needs_fitting* | Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances. |
| *parameters* | Returns the parameters which were used to initialize the component. |
| *predict* | Make predictions using fitted Prophet regressor. |
| *predict_proba* | Make probability estimates for labels. |
| *save* | Saves component at file path. |
| *update_parameters* | Updates the parameter dictionary of the component. |

**static build_prophet_df**(*X: pandas.DataFrame*, *y: Optional[pandas.Series] = None*, *time_index: str = 'ds'*) → pandas.DataFrame

Build the Prophet data to pass fit and predict on.

**clone**(*self*)

Constructs a new component with the same parameters and random state.

**Returns** A new instance of this component with identical parameters and random state.

**default_parameters**(*cls*) → dict

Returns the default parameters for this component.

**Returns** Default parameters for this component.

**Return type** dict

**describe**(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

**Parameters**

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description as dictionary in the format {"name": name, "parameters": parameters}

**Returns** Returns dictionary if return_dict is True, else None.

**Return type** None or dict

**property feature_importance**(*self*) → numpy.ndarray

Returns array of 0's with len(1) as feature_importance is not defined for Prophet regressor.

**fit**(*self*, *X: pandas.DataFrame*, *y: Optional[pandas.Series] = None*)

Fits Prophet regressor component to data.

**Parameters**

- **X** (*pd.DataFrame*) – The input training data of shape [n_samples, n_features].
- **y** (*pd.Series*) – The target training data of length [n_samples].

**Returns** self

**get_params**(*self*) → dict

Get parameters for the Prophet regressor.

**get_prediction_intervals**(*self*, *X: pandas.DataFrame*, *y: Optional[pandas.Series] = None*, *coverage: List[float] = None*, *predictions: pandas.Series = None*) → Dict[str, pandas.Series]

Find the prediction intervals using the fitted ProphetRegressor.

**Parameters**

- **X** (*pd.DataFrame*) – Data of shape [n_samples, n_features].
- **y** (*pd.Series*) – Target data. Ignored.
- **coverage** (*List[float]*) – A list of floats between the values 0 and 1 that the upper and lower bounds of the prediction interval should be calculated for.
- **predictions** (*pd.Series*) – Not used for Prophet estimator.

**Returns** Prediction intervals, keys are in the format {coverage}_lower or {coverage}_upper.

**Return type** dict

**static load**(*file_path*)

Loads component at file path.

**Parameters** **file_path** (*str*) – Location to load file.

**Returns** ComponentBase object

**needs_fitting**(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need to be fit or whose fit methods do nothing.

**Returns** True.

**property parameters**(*self*)

Returns the parameters which were used to initialize the component.

**predict**(*self*, *X: pandas.DataFrame*, *y: Optional[pandas.Series] = None*) → pandas.Series

Make predictions using fitted Prophet regressor.

**Parameters**

- **X** (*pd.DataFrame*) – Data of shape [n_samples, n_features].
- **y** (*pd.Series*) – Target data. Ignored.

**Returns** Predicted values.

**Return type** pd.Series

**predict_proba**(*self*, *X: pandas.DataFrame*) → pandas.Series

Make probability estimates for labels.

**Parameters** **X** (*pd.DataFrame*) – Features.

**Returns** Probability estimates.

**Return type** pd.Series

**Raises** **MethodPropertyNotFoundError** – If estimator does not have a predict_proba method or a component_obj that implements predict_proba.

**save**(*self*, *file_path*, *pickle_protocol=cloudpickle.DEFAULT_PROTOCOL*)

Saves component at file path.

**Parameters**

- **file_path** (*str*) – Location to save file.
- **pickle_protocol** (*int*) – The pickle data stream format.

**update_parameters**(*self*, *update_dict*, *reset_fit=True*)

Updates the parameter dictionary of the component.

**Parameters**

- **update_dict** (*dict*) – A dict of parameters to update.
- **reset_fit** (*bool, optional*) – If True, will set *_is_fitted* to False.

**class** evalml.pipelines.components.estimators.**RandomForestClassifier**(*n_estimators=100*, *max_depth=6*, *n_jobs=- 1*, *random_seed=0*, ***kwargs*)

Random Forest Classifier.

**Parameters**

- **n_estimators** (*float*) – The number of trees in the forest. Defaults to 100.
- **max_depth** (*int*) – Maximum tree depth for base learners. Defaults to 6.
- **n_jobs** (*int or None*) – Number of jobs to run in parallel. -1 uses all processes. Defaults to -1.
- **random_seed** (*int*) – Seed for the random number generator. Defaults to 0.

**Attributes**

| | |
|---|---|
| **hyper-parameter_ranges** | { "n_estimators": Integer(10, 1000), "max_depth": Integer(1, 10),} |
| **model_family** | ModelFamily.RANDOM_FOREST |
| **modifies_features** | True |
| **modifies_target** | False |
| **name** | Random Forest Classifier |
| **supported_problem_types** | [ ProblemTypes.BINARY, ProblemTypes.MULTICLASS, ProblemTypes.TIME_SERIES_BINARY, ProblemTypes.TIME_SERIES_MULTICLASS,] |
| **training_only** | False |

**Methods**

| | |
|---|---|
| *clone* | Constructs a new component with the same parameters and random state. |
| *default_parameters* | Returns the default parameters for this component. |
| *describe* | Describe a component and its parameters. |
| *feature_importance* | Returns importance associated with each feature. |
| *fit* | Fits estimator to data. |
| *get_prediction_intervals* | Find the prediction intervals using the fitted regressor. |
| *load* | Loads component at file path. |
| *needs_fitting* | Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances. |
| *parameters* | Returns the parameters which were used to initialize the component. |
| *predict* | Make predictions using selected features. |
| *predict_proba* | Make probability estimates for labels. |
| *save* | Saves component at file path. |
| *update_parameters* | Updates the parameter dictionary of the component. |

**clone**(*self*)

Constructs a new component with the same parameters and random state.

**Returns** A new instance of this component with identical parameters and random state.

**default_parameters**(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

**Returns** Default parameters for this component.

**Return type** dict

**describe**(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

**Parameters**

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description as dictionary in the format {"name": name, "parameters": parameters}

**Returns** Returns dictionary if return_dict is True, else None.

**Return type** None or dict

**property feature_importance**(*self*) → pandas.Series

Returns importance associated with each feature.

**Returns** Importance associated with each feature.

**Return type** np.ndarray

**Raises** **MethodPropertyNotFoundError** – If estimator does not have a feature_importance method or a component_obj that implements feature_importance.

**fit**(*self*, *X: pandas.DataFrame*, *y: Optional[pandas.Series] = None*)

Fits estimator to data.

**Parameters**

- **X** (*pd.DataFrame*) – The input training data of shape [n_samples, n_features].
- **y** (*pd.Series, optional*) – The target training data of length [n_samples].

**Returns** self

**get_prediction_intervals**(*self*, *X: pandas.DataFrame*, *y: Optional[pandas.Series] = None*, *coverage: List[float] = None*, *predictions: pandas.Series = None*) → Dict[str, pandas.Series]

Find the prediction intervals using the fitted regressor.

This function takes the predictions of the fitted estimator and calculates the rolling standard deviation across all predictions using a window size of 5. The lower and upper predictions are determined by taking the percent point (quantile) function of the lower tail probability at each bound multiplied by the rolling standard deviation.

**Parameters**

- **X** (*pd.DataFrame*) – Data of shape [n_samples, n_features].
- **y** (*pd.Series*) – Target data. Ignored.
- **coverage** (*list[float]*) – A list of floats between the values 0 and 1 that the upper and lower bounds of the prediction interval should be calculated for.
- **predictions** (*pd.Series*) – Optional list of predictions to use. If None, will generate predictions using *X*.

**Returns** Prediction intervals, keys are in the format {coverage}_lower or {coverage}_upper.

**Return type** dict

**Raises** **MethodPropertyNotFoundError** – If the estimator does not support Time Series Regression as a problem type.

**static load**(*file_path*)

Loads component at file path.

**Parameters** **file_path** (*str*) – Location to load file.

**Returns** ComponentBase object

**needs_fitting**(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need to be fit or whose fit methods do nothing.

**Returns** True.

**property parameters**(*self*)

Returns the parameters which were used to initialize the component.

**predict**(*self*, *X: pandas.DataFrame*) → pandas.Series

Make predictions using selected features.

**Parameters** **X** (*pd.DataFrame*) – Data of shape [n_samples, n_features].

**Returns** Predicted values.

**Return type** pd.Series

**Raises** `MethodPropertyNotFoundError` – If estimator does not have a predict method or a component_obj that implements predict.

`predict_proba`(*self*, *X: pandas.DataFrame*) → pandas.Series

Make probability estimates for labels.

**Parameters** `X` (*pd.DataFrame*) – Features.

**Returns** Probability estimates.

**Return type** pd.Series

**Raises** `MethodPropertyNotFoundError` – If estimator does not have a predict_proba method or a component_obj that implements predict_proba.

`save`(*self*, *file_path*, *pickle_protocol=cloudpickle.DEFAULT_PROTOCOL*)

Saves component at file path.

**Parameters**

- `file_path` (*str*) – Location to save file.
- `pickle_protocol` (*int*) – The pickle data stream format.

`update_parameters`(*self*, *update_dict*, *reset_fit=True*)

Updates the parameter dictionary of the component.

**Parameters**

- `update_dict` (*dict*) – A dict of parameters to update.
- `reset_fit` (*bool, optional*) – If True, will set *_is_fitted* to False.

`class` `evalml.pipelines.components.estimators.`**`RandomForestRegressor`**(*n_estimators: int = 100*, *max_depth: int = 6*, *n_jobs: int = - 1*, *random_seed: Union[int, float] = 0*, ***kwargs*)

Random Forest Regressor.

**Parameters**

- `n_estimators` (*float*) – The number of trees in the forest. Defaults to 100.
- `max_depth` (*int*) – Maximum tree depth for base learners. Defaults to 6.
- `n_jobs` (*int or None*) – Number of jobs to run in parallel. -1 uses all processes. Defaults to -1.
- `random_seed` (*int*) – Seed for the random number generator. Defaults to 0.

**Attributes**

| | |
|---|---|
| **hyper-parameter_ranges** | { "n_estimators": Integer(10, 1000), "max_depth": Integer(1, 32),} |
| **model_family** | ModelFamily.RANDOM_FOREST |
| **modifies_features** | True |
| **modifies_target** | False |
| **name** | Random Forest Regressor |
| **supported_problem_types** | [ ProblemTypes.REGRESSION, ProblemTypes.TIME_SERIES_REGRESSION,] |
| **training_only** | False |

**Methods**

| | |
|---|---|
| *clone* | Constructs a new component with the same parameters and random state. |
| *default_parameters* | Returns the default parameters for this component. |
| *describe* | Describe a component and its parameters. |
| *feature_importance* | Returns importance associated with each feature. |
| *fit* | Fits estimator to data. |
| *get_prediction_intervals* | Find the prediction intervals using the fitted RandomForestRegressor. |
| *load* | Loads component at file path. |
| *needs_fitting* | Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances. |
| *parameters* | Returns the parameters which were used to initialize the component. |
| *predict* | Make predictions using selected features. |
| *predict_proba* | Make probability estimates for labels. |
| *save* | Saves component at file path. |
| *update_parameters* | Updates the parameter dictionary of the component. |

**clone**(*self*)

Constructs a new component with the same parameters and random state.

**Returns** A new instance of this component with identical parameters and random state.

**default_parameters**(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

**Returns** Default parameters for this component.

**Return type** dict

**describe**(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

**Parameters**

- **print_name** (*bool, optional*) – whether to print name of component

- **return_dict** (*bool, optional*) – whether to return description as dictionary in the format {"name": name, "parameters": parameters}

**Returns** Returns dictionary if return_dict is True, else None.

**Return type** None or dict

**property feature_importance**(*self*) → pandas.Series

Returns importance associated with each feature.

**Returns** Importance associated with each feature.

**Return type** np.ndarray

**Raises MethodPropertyNotFoundError** – If estimator does not have a feature_importance method or a component_obj that implements feature_importance.

**fit**(*self, X: pandas.DataFrame, y: Optional[pandas.Series] = None*)

Fits estimator to data.

**Parameters**

- **X** (*pd.DataFrame*) – The input training data of shape [n_samples, n_features].
- **y** (*pd.Series, optional*) – The target training data of length [n_samples].

**Returns** self

**get_prediction_intervals**(*self, X: pandas.DataFrame, y: Optional[pandas.Series] = None, coverage: List[float] = None, predictions: pandas.Series = None*) → Dict[str, pandas.Series]

Find the prediction intervals using the fitted RandomForestRegressor.

**Parameters**

- **X** (*pd.DataFrame*) – Data of shape [n_samples, n_features].
- **y** (*pd.Series*) – Target data. Optional.
- **coverage** (*list[float]*) – A list of floats between the values 0 and 1 that the upper and lower bounds of the prediction interval should be calculated for.
- **predictions** (*pd.Series*) – Optional list of predictions to use. If None, will generate predictions using *X*.

**Returns** Prediction intervals, keys are in the format {coverage}_lower or {coverage}_upper.

**Return type** dict

**static load**(*file_path*)

Loads component at file path.

**Parameters file_path** (*str*) – Location to load file.

**Returns** ComponentBase object

**needs_fitting**(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need to be fit or whose fit methods do nothing.

**Returns** True.

**property parameters**(*self*)

Returns the parameters which were used to initialize the component.

**predict**(*self*, *X: pandas.DataFrame*) → pandas.Series

Make predictions using selected features.

**Parameters** **X** (*pd.DataFrame*) – Data of shape [n_samples, n_features].

**Returns** Predicted values.

**Return type** pd.Series

**Raises** **MethodPropertyNotFoundError** – If estimator does not have a predict method or a component_obj that implements predict.

**predict_proba**(*self*, *X: pandas.DataFrame*) → pandas.Series

Make probability estimates for labels.

**Parameters** **X** (*pd.DataFrame*) – Features.

**Returns** Probability estimates.

**Return type** pd.Series

**Raises** **MethodPropertyNotFoundError** – If estimator does not have a predict_proba method or a component_obj that implements predict_proba.

**save**(*self*, *file_path*, *pickle_protocol=cloudpickle.DEFAULT_PROTOCOL*)

Saves component at file path.

**Parameters**

- **file_path** (*str*) – Location to save file.
- **pickle_protocol** (*int*) – The pickle data stream format.

**update_parameters**(*self*, *update_dict*, *reset_fit=True*)

Updates the parameter dictionary of the component.

**Parameters**

- **update_dict** (*dict*) – A dict of parameters to update.
- **reset_fit** (*bool, optional*) – If True, will set *_is_fitted* to False.

**class** evalml.pipelines.components.estimators.**SVMClassifier**(*C=1.0*, *kernel='rbf'*, *gamma='auto'*, *probability=True*, *random_seed=0*, ***kwargs*)

Support Vector Machine Classifier.

**Parameters**

- **C** (*float*) – The regularization parameter. The strength of the regularization is inversely proportional to C. Must be strictly positive. The penalty is a squared l2 penalty. Defaults to 1.0.
- **kernel** (*{"poly", "rbf", "sigmoid"}*) – Specifies the kernel type to be used in the algorithm. Defaults to "rbf".
- **gamma** (*{"scale", "auto"} or float*) – Kernel coefficient for "rbf", "poly" and "sigmoid". Defaults to "auto". - If gamma='scale' is passed then it uses 1 / (n_features * X.var()) as value of gamma - If "auto" (default), uses 1 / n_features
- **probability** (*boolean*) – Whether to enable probability estimates. Defaults to True.

- **random_seed** (*int*) – Seed for the random number generator. Defaults to 0.

**Attributes**

| | |
|---|---|
| **hyper-parameter_ranges** | { "C": Real(0, 10), "kernel": ["poly", "rbf", "sigmoid"], "gamma": ["scale", "auto"],} |
| **model_family** | ModelFamily.SVM |
| **modifies_features** | True |
| **modifies_target** | False |
| **name** | SVM Classifier |
| **supported_problem_types** | [ ProblemTypes.BINARY, ProblemTypes.MULTICLASS, ProblemTypes.TIME_SERIES_BINARY, ProblemTypes.TIME_SERIES_MULTICLASS,] |
| **training_only** | False |

**Methods**

| | |
|---|---|
| *clone* | Constructs a new component with the same parameters and random state. |
| *default_parameters* | Returns the default parameters for this component. |
| *describe* | Describe a component and its parameters. |
| *feature_importance* | Feature importance only works with linear kernels. |
| *fit* | Fits estimator to data. |
| *get_prediction_intervals* | Find the prediction intervals using the fitted regressor. |
| *load* | Loads component at file path. |
| *needs_fitting* | Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances. |
| *parameters* | Returns the parameters which were used to initialize the component. |
| *predict* | Make predictions using selected features. |
| *predict_proba* | Make probability estimates for labels. |
| *save* | Saves component at file path. |
| *update_parameters* | Updates the parameter dictionary of the component. |

**clone**(*self*)

Constructs a new component with the same parameters and random state.

**Returns** A new instance of this component with identical parameters and random state.

**default_parameters**(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

**Returns** Default parameters for this component.

**Return type** dict

**describe**(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

**Parameters**

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description as dictionary in the format {"name": name, "parameters": parameters}

**Returns** Returns dictionary if return_dict is True, else None.

**Return type** None or dict

**property feature_importance**(*self*)

Feature importance only works with linear kernels.

If the kernel isn't linear, we return a numpy array of zeros.

**Returns** Feature importance of fitted SVM classifier or a numpy array of zeroes if the kernel is not linear.

**fit**(*self, X: pandas.DataFrame, y: Optional[pandas.Series] = None*)

Fits estimator to data.

**Parameters**

- **X** (*pd.DataFrame*) – The input training data of shape [n_samples, n_features].
- **y** (*pd.Series, optional*) – The target training data of length [n_samples].

**Returns** self

**get_prediction_intervals**(*self, X: pandas.DataFrame, y: Optional[pandas.Series] = None, coverage: List[float] = None, predictions: pandas.Series = None*) → Dict[str, pandas.Series]

Find the prediction intervals using the fitted regressor.

This function takes the predictions of the fitted estimator and calculates the rolling standard deviation across all predictions using a window size of 5. The lower and upper predictions are determined by taking the percent point (quantile) function of the lower tail probability at each bound multiplied by the rolling standard deviation.

**Parameters**

- **X** (*pd.DataFrame*) – Data of shape [n_samples, n_features].
- **y** (*pd.Series*) – Target data. Ignored.
- **coverage** (*list[float]*) – A list of floats between the values 0 and 1 that the upper and lower bounds of the prediction interval should be calculated for.
- **predictions** (*pd.Series*) – Optional list of predictions to use. If None, will generate predictions using *X*.

**Returns** Prediction intervals, keys are in the format {coverage}_lower or {coverage}_upper.

**Return type** dict

**Raises** **MethodPropertyNotFoundError** – If the estimator does not support Time Series Regression as a problem type.

**static load**(*file_path*)

Loads component at file path.

**Parameters** **file_path** (*str*) – Location to load file.

**Returns** ComponentBase object

**needs_fitting**(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need to be fit or whose fit methods do nothing.

**Returns** True.

**property parameters**(*self*)

Returns the parameters which were used to initialize the component.

**predict**(*self*, *X: pandas.DataFrame*) → pandas.Series

Make predictions using selected features.

**Parameters** **X** (*pd.DataFrame*) – Data of shape [n_samples, n_features].

**Returns** Predicted values.

**Return type** pd.Series

**Raises** **MethodPropertyNotFoundError** – If estimator does not have a predict method or a component_obj that implements predict.

**predict_proba**(*self*, *X: pandas.DataFrame*) → pandas.Series

Make probability estimates for labels.

**Parameters** **X** (*pd.DataFrame*) – Features.

**Returns** Probability estimates.

**Return type** pd.Series

**Raises** **MethodPropertyNotFoundError** – If estimator does not have a predict_proba method or a component_obj that implements predict_proba.

**save**(*self*, *file_path*, *pickle_protocol=cloudpickle.DEFAULT_PROTOCOL*)

Saves component at file path.

**Parameters**

- **file_path** (*str*) – Location to save file.
- **pickle_protocol** (*int*) – The pickle data stream format.

**update_parameters**(*self*, *update_dict*, *reset_fit=True*)

Updates the parameter dictionary of the component.

**Parameters**

- **update_dict** (*dict*) – A dict of parameters to update.
- **reset_fit** (*bool, optional*) – If True, will set *_is_fitted* to False.

**class** evalml.pipelines.components.estimators.**SVMRegressor**(*C=1.0*, *kernel='rbf'*, *gamma='auto'*, *random_seed=0*, ***kwargs*)

Support Vector Machine Regressor.

**Parameters**

- **C** (*float*) – The regularization parameter. The strength of the regularization is inversely proportional to C. Must be strictly positive. The penalty is a squared l2 penalty. Defaults to 1.0.
- **kernel** (*{"poly", "rbf", "sigmoid"}*) – Specifies the kernel type to be used in the algorithm. Defaults to "rbf".

- **gamma** (*{"scale", "auto"} or float*) – Kernel coefficient for "rbf", "poly" and "sigmoid". Defaults to "auto". - If gamma='scale' is passed then it uses 1 / (n_features * X.var()) as value of gamma - If "auto" (default), uses 1 / n_features
- **random_seed** (*int*) – Seed for the random number generator. Defaults to 0.

**Attributes**

| | |
|---|---|
| **hyperparameter_ranges** | { "C": Real(0, 10), "kernel": ["poly", "rbf", "sigmoid"], "gamma": ["scale", "auto"],} |
| **model_family** | ModelFamily.SVM |
| **modifies_features** | True |
| **modifies_target** | False |
| **name** | SVM Regressor |
| **supported_problem_types** | [ ProblemTypes.REGRESSION, ProblemTypes.TIME_SERIES_REGRESSION,] |
| **training_only** | False |

**Methods**

| | |
|---|---|
| *clone* | Constructs a new component with the same parameters and random state. |
| *default_parameters* | Returns the default parameters for this component. |
| *describe* | Describe a component and its parameters. |
| *feature_importance* | Feature importance of fitted SVM regresor. |
| *fit* | Fits estimator to data. |
| *get_prediction_intervals* | Find the prediction intervals using the fitted regressor. |
| *load* | Loads component at file path. |
| *needs_fitting* | Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances. |
| *parameters* | Returns the parameters which were used to initialize the component. |
| *predict* | Make predictions using selected features. |
| *predict_proba* | Make probability estimates for labels. |
| *save* | Saves component at file path. |
| *update_parameters* | Updates the parameter dictionary of the component. |

**clone**(*self*)

Constructs a new component with the same parameters and random state.

**Returns** A new instance of this component with identical parameters and random state.

**default_parameters**(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

**Returns** Default parameters for this component.

**Return type** dict

**describe**(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

**Parameters**

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description as dictionary in the format {"name": name, "parameters": parameters}

**Returns** Returns dictionary if return_dict is True, else None.

**Return type** None or dict

**property feature_importance**(*self*)

Feature importance of fitted SVM regresor.

Only works with linear kernels. If the kernel isn't linear, we return a numpy array of zeros.

**Returns** The feature importance of the fitted SVM regressor, or an array of zeroes if the kernel is not linear.

**fit**(*self*, *X: pandas.DataFrame*, *y: Optional[pandas.Series] = None*)

Fits estimator to data.

**Parameters**

- **X** (*pd.DataFrame*) – The input training data of shape [n_samples, n_features].
- **y** (*pd.Series, optional*) – The target training data of length [n_samples].

**Returns** self

**get_prediction_intervals**(*self*, *X: pandas.DataFrame*, *y: Optional[pandas.Series] = None*, *coverage: List[float] = None*, *predictions: pandas.Series = None*) → Dict[str, pandas.Series]

Find the prediction intervals using the fitted regressor.

This function takes the predictions of the fitted estimator and calculates the rolling standard deviation across all predictions using a window size of 5. The lower and upper predictions are determined by taking the percent point (quantile) function of the lower tail probability at each bound multiplied by the rolling standard deviation.

**Parameters**

- **X** (*pd.DataFrame*) – Data of shape [n_samples, n_features].
- **y** (*pd.Series*) – Target data. Ignored.
- **coverage** (*list[float]*) – A list of floats between the values 0 and 1 that the upper and lower bounds of the prediction interval should be calculated for.
- **predictions** (*pd.Series*) – Optional list of predictions to use. If None, will generate predictions using *X*.

**Returns** Prediction intervals, keys are in the format {coverage}_lower or {coverage}_upper.

**Return type** dict

**Raises** **MethodPropertyNotFoundError** – If the estimator does not support Time Series Regression as a problem type.

**static load**(*file_path*)

Loads component at file path.

**Parameters** **file_path** (*str*) – Location to load file.

**Returns** ComponentBase object

**needs_fitting**(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need to be fit or whose fit methods do nothing.

**Returns** True.

**property parameters**(*self*)

Returns the parameters which were used to initialize the component.

**predict**(*self*, *X: pandas.DataFrame*) → pandas.Series

Make predictions using selected features.

**Parameters** **X** (*pd.DataFrame*) – Data of shape [n_samples, n_features].

**Returns** Predicted values.

**Return type** pd.Series

**Raises** **MethodPropertyNotFoundError** – If estimator does not have a predict method or a component_obj that implements predict.

**predict_proba**(*self*, *X: pandas.DataFrame*) → pandas.Series

Make probability estimates for labels.

**Parameters** **X** (*pd.DataFrame*) – Features.

**Returns** Probability estimates.

**Return type** pd.Series

**Raises** **MethodPropertyNotFoundError** – If estimator does not have a predict_proba method or a component_obj that implements predict_proba.

**save**(*self*, *file_path*, *pickle_protocol=cloudpickle.DEFAULT_PROTOCOL*)

Saves component at file path.

**Parameters**

- **file_path** (*str*) – Location to save file.
- **pickle_protocol** (*int*) – The pickle data stream format.

**update_parameters**(*self*, *update_dict*, *reset_fit=True*)

Updates the parameter dictionary of the component.

**Parameters**

- **update_dict** (*dict*) – A dict of parameters to update.
- **reset_fit** (*bool, optional*) – If True, will set *_is_fitted* to False.

**class** evalml.pipelines.components.estimators.**TimeSeriesBaselineEstimator**(*gap=1*, *forecast_horizon=1*, *random_seed=0*, ***kwargs*)

Time series estimator that predicts using the naive forecasting approach.

This is useful as a simple baseline estimator for time series problems.

**Parameters**

- **gap** (*int*) – Gap between prediction date and target date and must be a positive integer. If gap is 0, target date will be shifted ahead by 1 time period. Defaults to 1.
- **forecast_horizon** (*int*) – Number of time steps the model is expected to predict.
- **random_seed** (*int*) – Seed for the random number generator. Defaults to 0.

**Attributes**

| | |
|---|---|
| **hyper-parameter_ranges** | {} |
| **model_family** | ModelFamily.BASELINE |
| **modifies_features** | True |
| **modifies_target** | False |
| **name** | Time Series Baseline Estimator |
| **supported_problem_types** | [ ProblemTypes.TIME_SERIES_REGRESSION, ProblemTypes.TIME_SERIES_BINARY, ProblemTypes.TIME_SERIES_MULTICLASS,] |
| **training_only** | False |

**Methods**

| | |
|---|---|
| *clone* | Constructs a new component with the same parameters and random state. |
| *default_parameters* | Returns the default parameters for this component. |
| *describe* | Describe a component and its parameters. |
| *feature_importance* | Returns importance associated with each feature. |
| *fit* | Fits time series baseline estimator to data. |
| *get_prediction_intervals* | Find the prediction intervals using the fitted regressor. |
| *load* | Loads component at file path. |
| *needs_fitting* | Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances. |
| *parameters* | Returns the parameters which were used to initialize the component. |
| *predict* | Make predictions using fitted time series baseline estimator. |
| *predict_proba* | Make prediction probabilities using fitted time series baseline estimator. |
| *save* | Saves component at file path. |
| *update_parameters* | Updates the parameter dictionary of the component. |

**clone**(*self*)

Constructs a new component with the same parameters and random state.

**Returns** A new instance of this component with identical parameters and random state.

**default_parameters**(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

**Returns** Default parameters for this component.

**Return type** dict

**describe**(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

**Parameters**

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description as dictionary in the format {"name": name, "parameters": parameters}

**Returns** Returns dictionary if return_dict is True, else None.

**Return type** None or dict

**property feature_importance**(*self*)

Returns importance associated with each feature.

Since baseline estimators do not use input features to calculate predictions, returns an array of zeroes.

**Returns** An array of zeroes.

**Return type** np.ndarray (float)

**fit**(*self*, *X*, *y=None*)

Fits time series baseline estimator to data.

**Parameters**

- **X** (*pd.DataFrame*) – The input training data of shape [n_samples, n_features].
- **y** (*pd.Series*) – The target training data of length [n_samples].

**Returns** self

**Raises** **ValueError** – If input y is None.

**get_prediction_intervals**(*self*, *X: pandas.DataFrame*, *y: Optional[pandas.Series] = None*, *coverage: List[float] = None*, *predictions: pandas.Series = None*) → Dict[str, pandas.Series]

Find the prediction intervals using the fitted regressor.

This function takes the predictions of the fitted estimator and calculates the rolling standard deviation across all predictions using a window size of 5. The lower and upper predictions are determined by taking the percent point (quantile) function of the lower tail probability at each bound multiplied by the rolling standard deviation.

**Parameters**

- **X** (*pd.DataFrame*) – Data of shape [n_samples, n_features].
- **y** (*pd.Series*) – Target data. Ignored.
- **coverage** (*list[float]*) – A list of floats between the values 0 and 1 that the upper and lower bounds of the prediction interval should be calculated for.

- **predictions** (*pd.Series*) – Optional list of predictions to use. If None, will generate predictions using *X*.

**Returns** Prediction intervals, keys are in the format {coverage}_lower or {coverage}_upper.

**Return type** dict

**Raises** **MethodPropertyNotFoundError** – If the estimator does not support Time Series Regression as a problem type.

**static load**(*file_path*)

Loads component at file path.

**Parameters** **file_path** (*str*) – Location to load file.

**Returns** ComponentBase object

**needs_fitting**(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need to be fit or whose fit methods do nothing.

**Returns** True.

**property parameters**(*self*)

Returns the parameters which were used to initialize the component.

**predict**(*self*, *X*)

Make predictions using fitted time series baseline estimator.

**Parameters** **X** (*pd.DataFrame*) – Data of shape [n_samples, n_features].

**Returns** Predicted values.

**Return type** pd.Series

**Raises** **ValueError** – If input y is None.

**predict_proba**(*self*, *X*)

Make prediction probabilities using fitted time series baseline estimator.

**Parameters** **X** (*pd.DataFrame*) – Data of shape [n_samples, n_features].

**Returns** Predicted probability values.

**Return type** pd.DataFrame

**Raises** **ValueError** – If input y is None.

**save**(*self*, *file_path*, *pickle_protocol=cloudpickle.DEFAULT_PROTOCOL*)

Saves component at file path.

**Parameters**

- **file_path** (*str*) – Location to save file.
- **pickle_protocol** (*int*) – The pickle data stream format.

**update_parameters**(*self*, *update_dict*, *reset_fit=True*)

Updates the parameter dictionary of the component.

**Parameters**

- **update_dict** (*dict*) – A dict of parameters to update.

- **reset_fit** (*bool, optional*) – If True, will set *_is_fitted* to False.

**class** evalml.pipelines.components.estimators.**VowpalWabbitBinaryClassifier**(*loss_function='logistic', learning_rate=0.5, decay_learning_rate=1.0, power_t=0.5, passes=1, random_seed=0, **kwargs*)

Vowpal Wabbit Binary Classifier.

**Parameters**

- **loss_function** (*str*) – Specifies the loss function to use. One of {"squared", "classic", "hinge", "logistic", "quantile"}. Defaults to "logistic".
- **learning_rate** (*float*) – Boosting learning rate. Defaults to 0.5.
- **decay_learning_rate** (*float*) – Decay factor for learning_rate. Defaults to 1.0.
- **power_t** (*float*) – Power on learning rate decay. Defaults to 0.5.
- **passes** (*int*) – Number of training passes. Defaults to 1.
- **random_seed** (*int*) – Seed for the random number generator. Defaults to 0.

**Attributes**

| | |
|---|---|
| **hyper-parameter_ranges** | None |
| **model_family** | ModelFamily.VOWPAL_WABBIT |
| **modifies_features** | True |
| **modifies_target** | False |
| **name** | Vowpal Wabbit Binary Classifier |
| **supported_problem_types** | [ ProblemTypes.BINARY, ProblemTypes.TIME_SERIES_BINARY,] |
| **training_only** | False |

**Methods**

| | |
|---|---|
| *clone* | Constructs a new component with the same parameters and random state. |
| *default_parameters* | Returns the default parameters for this component. |
| *describe* | Describe a component and its parameters. |
| *feature_importance* | Feature importance for Vowpal Wabbit classifiers. This is not implemented. |
| *fit* | Fits estimator to data. |
| *get_prediction_intervals* | Find the prediction intervals using the fitted regressor. |
| *load* | Loads component at file path. |
| *needs_fitting* | Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances. |
| *parameters* | Returns the parameters which were used to initialize the component. |
| *predict* | Make predictions using selected features. |
| *predict_proba* | Make probability estimates for labels. |
| *save* | Saves component at file path. |
| *update_parameters* | Updates the parameter dictionary of the component. |

**clone**(*self*)

Constructs a new component with the same parameters and random state.

**Returns** A new instance of this component with identical parameters and random state.

**default_parameters**(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

**Returns** Default parameters for this component.

**Return type** dict

**describe**(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

**Parameters**

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description as dictionary in the format {"name": name, "parameters": parameters}

**Returns** Returns dictionary if return_dict is True, else None.

**Return type** None or dict

**property feature_importance**(*self*)

Feature importance for Vowpal Wabbit classifiers. This is not implemented.

**fit**(*self*, *X: pandas.DataFrame*, *y: Optional[pandas.Series] = None*)

Fits estimator to data.

**Parameters**

- **X** (*pd.DataFrame*) – The input training data of shape [n_samples, n_features].
- **y** (*pd.Series, optional*) – The target training data of length [n_samples].

**Returns** self

**get_prediction_intervals**(*self*, *X: pandas.DataFrame*, *y: Optional[pandas.Series] = None*, *coverage: List[float] = None*, *predictions: pandas.Series = None*) → Dict[str, pandas.Series]

Find the prediction intervals using the fitted regressor.

This function takes the predictions of the fitted estimator and calculates the rolling standard deviation across all predictions using a window size of 5. The lower and upper predictions are determined by taking the percent point (quantile) function of the lower tail probability at each bound multiplied by the rolling standard deviation.

**Parameters**

- **X** (*pd.DataFrame*) – Data of shape [n_samples, n_features].
- **y** (*pd.Series*) – Target data. Ignored.
- **coverage** (*list[float]*) – A list of floats between the values 0 and 1 that the upper and lower bounds of the prediction interval should be calculated for.
- **predictions** (*pd.Series*) – Optional list of predictions to use. If None, will generate predictions using *X*.

**Returns** Prediction intervals, keys are in the format {coverage}_lower or {coverage}_upper.

**Return type** dict

**Raises** **MethodPropertyNotFoundError** – If the estimator does not support Time Series Regression as a problem type.

**static load**(*file_path*)

Loads component at file path.

**Parameters** **file_path** (*str*) – Location to load file.

**Returns** ComponentBase object

**needs_fitting**(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need to be fit or whose fit methods do nothing.

**Returns** True.

**property parameters**(*self*)

Returns the parameters which were used to initialize the component.

**predict**(*self*, *X: pandas.DataFrame*) → pandas.Series

Make predictions using selected features.

**Parameters** **X** (*pd.DataFrame*) – Data of shape [n_samples, n_features].

**Returns** Predicted values.

**Return type** pd.Series

**Raises** **MethodPropertyNotFoundError** – If estimator does not have a predict method or a component_obj that implements predict.

**predict_proba**(*self*, *X: pandas.DataFrame*) → pandas.Series

Make probability estimates for labels.

**Parameters** **X** (*pd.DataFrame*) – Features.

**Returns** Probability estimates.

**Return type** pd.Series

**Raises** **MethodPropertyNotFoundError** – If estimator does not have a predict_proba method or a component_obj that implements predict_proba.

**save**(*self*, *file_path*, *pickle_protocol=cloudpickle.DEFAULT_PROTOCOL*)

Saves component at file path.

**Parameters**

- **file_path** (*str*) – Location to save file.
- **pickle_protocol** (*int*) – The pickle data stream format.

**update_parameters**(*self*, *update_dict*, *reset_fit=True*)

Updates the parameter dictionary of the component.

**Parameters**

- **update_dict** (*dict*) – A dict of parameters to update.
- **reset_fit** (*bool, optional*) – If True, will set *_is_fitted* to False.

**class** evalml.pipelines.components.estimators.**VowpalWabbitMulticlassClassifier**(*loss_function='logistic', learning_rate=0.5, decay_learning_rate=1.0, power_t=0.5, passes=1, random_seed=0, **kwargs*)

Vowpal Wabbit Multiclass Classifier.

**Parameters**

- **loss_function** (*str*) – Specifies the loss function to use. One of {"squared", "classic", "hinge", "logistic", "quantile"}. Defaults to "logistic".
- **learning_rate** (*float*) – Boosting learning rate. Defaults to 0.5.
- **decay_learning_rate** (*float*) – Decay factor for learning_rate. Defaults to 1.0.
- **power_t** (*float*) – Power on learning rate decay. Defaults to 0.5.
- **random_seed** (*int*) – Seed for the random number generator. Defaults to 0.

**Attributes**

| | |
|---|---|
| **hyper-parameter_ranges** | None |
| **model_family** | ModelFamily.VOWPAL_WABBIT |
| **modifies_features** | True |
| **modifies_target** | False |
| **name** | Vowpal Wabbit Multiclass Classifier |
| **supported_problem_types** | [ ProblemTypes.MULTICLASS, ProblemTypes.TIME_SERIES_MULTICLASS,] |
| **training_only** | False |

**Methods**

| | |
|---|---|
| *clone* | Constructs a new component with the same parameters and random state. |
| *default_parameters* | Returns the default parameters for this component. |
| *describe* | Describe a component and its parameters. |
| *feature_importance* | Feature importance for Vowpal Wabbit classifiers. This is not implemented. |
| *fit* | Fits estimator to data. |
| *get_prediction_intervals* | Find the prediction intervals using the fitted regressor. |
| *load* | Loads component at file path. |
| *needs_fitting* | Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances. |
| *parameters* | Returns the parameters which were used to initialize the component. |
| *predict* | Make predictions using selected features. |
| *predict_proba* | Make probability estimates for labels. |
| *save* | Saves component at file path. |
| *update_parameters* | Updates the parameter dictionary of the component. |

**clone**(*self*)

Constructs a new component with the same parameters and random state.

**Returns** A new instance of this component with identical parameters and random state.

**default_parameters**(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

**Returns** Default parameters for this component.

**Return type** dict

**describe**(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

**Parameters**

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description as dictionary in the format {"name": name, "parameters": parameters}

**Returns** Returns dictionary if return_dict is True, else None.

**Return type** None or dict

**property feature_importance**(*self*)

Feature importance for Vowpal Wabbit classifiers. This is not implemented.

**fit**(*self*, *X: pandas.DataFrame*, *y: Optional[pandas.Series] = None*)

Fits estimator to data.

**Parameters**

- **X** (*pd.DataFrame*) – The input training data of shape [n_samples, n_features].
- **y** (*pd.Series, optional*) – The target training data of length [n_samples].

**Returns** self

**get_prediction_intervals**(*self*, *X: pandas.DataFrame*, *y: Optional[pandas.Series] = None*, *coverage: List[float] = None*, *predictions: pandas.Series = None*) → Dict[str, pandas.Series]

Find the prediction intervals using the fitted regressor.

This function takes the predictions of the fitted estimator and calculates the rolling standard deviation across all predictions using a window size of 5. The lower and upper predictions are determined by taking the percent point (quantile) function of the lower tail probability at each bound multiplied by the rolling standard deviation.

**Parameters**

- **X** (*pd.DataFrame*) – Data of shape [n_samples, n_features].
- **y** (*pd.Series*) – Target data. Ignored.
- **coverage** (*list[float]*) – A list of floats between the values 0 and 1 that the upper and lower bounds of the prediction interval should be calculated for.
- **predictions** (*pd.Series*) – Optional list of predictions to use. If None, will generate predictions using *X*.

**Returns** Prediction intervals, keys are in the format {coverage}_lower or {coverage}_upper.

**Return type** dict

**Raises** **MethodPropertyNotFoundError** – If the estimator does not support Time Series Regression as a problem type.

**static load**(*file_path*)

Loads component at file path.

**Parameters** **file_path** (*str*) – Location to load file.

**Returns** ComponentBase object

**needs_fitting**(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need to be fit or whose fit methods do nothing.

**Returns** True.

**property parameters**(*self*)

Returns the parameters which were used to initialize the component.

**predict**(*self*, *X: pandas.DataFrame*) → pandas.Series

Make predictions using selected features.

**Parameters** **X** (*pd.DataFrame*) – Data of shape [n_samples, n_features].

**Returns** Predicted values.

**Return type** pd.Series

**Raises** **MethodPropertyNotFoundError** – If estimator does not have a predict method or a component_obj that implements predict.

**predict_proba**(*self*, *X: pandas.DataFrame*) → pandas.Series

Make probability estimates for labels.

**Parameters** **X** (*pd.DataFrame*) – Features.

**Returns** Probability estimates.

**Return type** pd.Series

**Raises** **MethodPropertyNotFoundError** – If estimator does not have a predict_proba method or a component_obj that implements predict_proba.

**save**(*self*, *file_path*, *pickle_protocol=cloudpickle.DEFAULT_PROTOCOL*)

Saves component at file path.

**Parameters**

- **file_path** (*str*) – Location to save file.
- **pickle_protocol** (*int*) – The pickle data stream format.

**update_parameters**(*self*, *update_dict*, *reset_fit=True*)

Updates the parameter dictionary of the component.

**Parameters**

- **update_dict** (*dict*) – A dict of parameters to update.
- **reset_fit** (*bool, optional*) – If True, will set *_is_fitted* to False.

**class** evalml.pipelines.components.estimators.**VowpalWabbitRegressor**(*learning_rate=0.5*, *decay_learning_rate=1.0*, *power_t=0.5*, *passes=1*, *random_seed=0*, ***kwargs*)

Vowpal Wabbit Regressor.

**Parameters**

- **learning_rate** (*float*) – Boosting learning rate. Defaults to 0.5.
- **decay_learning_rate** (*float*) – Decay factor for learning_rate. Defaults to 1.0.
- **power_t** (*float*) – Power on learning rate decay. Defaults to 0.5.
- **passes** (*int*) – Number of training passes. Defaults to 1.
- **random_seed** (*int*) – Seed for the random number generator. Defaults to 0.

**Attributes**

| | |
|---|---|
| **hyper-parameter_ranges** | None |
| **model_family** | ModelFamily.VOWPAL_WABBIT |
| **modifies_features** | True |
| **modifies_target** | False |
| **name** | Vowpal Wabbit Regressor |
| **supported_problem_types** | [ ProblemTypes.REGRESSION, ProblemTypes.TIME_SERIES_REGRESSION,] |
| **training_only** | False |

**Methods**

| | |
|---|---|
| *clone* | Constructs a new component with the same parameters and random state. |
| *default_parameters* | Returns the default parameters for this component. |
| *describe* | Describe a component and its parameters. |
| *feature_importance* | Feature importance for Vowpal Wabbit regressor. |
| *fit* | Fits estimator to data. |
| *get_prediction_intervals* | Find the prediction intervals using the fitted regressor. |
| *load* | Loads component at file path. |
| *needs_fitting* | Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances. |
| *parameters* | Returns the parameters which were used to initialize the component. |
| *predict* | Make predictions using selected features. |
| *predict_proba* | Make probability estimates for labels. |
| *save* | Saves component at file path. |
| *update_parameters* | Updates the parameter dictionary of the component. |

`clone`(*self*)

Constructs a new component with the same parameters and random state.

**Returns** A new instance of this component with identical parameters and random state.

`default_parameters`(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

**Returns** Default parameters for this component.

**Return type** dict

`describe`(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

**Parameters**

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description as dictionary in the format {"name": name, "parameters": parameters}

**Returns** Returns dictionary if return_dict is True, else None.

**Return type** None or dict

**property feature_importance**(*self*)

Feature importance for Vowpal Wabbit regressor.

**fit**(*self, X: pandas.DataFrame, y: Optional[pandas.Series] = None*)

Fits estimator to data.

**Parameters**

- **X** (*pd.DataFrame*) – The input training data of shape [n_samples, n_features].
- **y** (*pd.Series, optional*) – The target training data of length [n_samples].

**Returns** self

**get_prediction_intervals**(*self, X: pandas.DataFrame, y: Optional[pandas.Series] = None, coverage: List[float] = None, predictions: pandas.Series = None*) → Dict[str, pandas.Series]

Find the prediction intervals using the fitted regressor.

This function takes the predictions of the fitted estimator and calculates the rolling standard deviation across all predictions using a window size of 5. The lower and upper predictions are determined by taking the percent point (quantile) function of the lower tail probability at each bound multiplied by the rolling standard deviation.

**Parameters**

- **X** (*pd.DataFrame*) – Data of shape [n_samples, n_features].
- **y** (*pd.Series*) – Target data. Ignored.
- **coverage** (*list[float]*) – A list of floats between the values 0 and 1 that the upper and lower bounds of the prediction interval should be calculated for.
- **predictions** (*pd.Series*) – Optional list of predictions to use. If None, will generate predictions using *X*.

**Returns** Prediction intervals, keys are in the format {coverage}_lower or {coverage}_upper.

**Return type** dict

**Raises** **MethodPropertyNotFoundError** – If the estimator does not support Time Series Regression as a problem type.

**static load**(*file_path*)

Loads component at file path.

**Parameters** **file_path** (*str*) – Location to load file.

**Returns** ComponentBase object

**needs_fitting**(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need to be fit or whose fit methods do nothing.

**Returns** True.

**property parameters**(*self*)

Returns the parameters which were used to initialize the component.

**predict**(*self*, *X: pandas.DataFrame*) → pandas.Series

Make predictions using selected features.

**Parameters** **X** (*pd.DataFrame*) – Data of shape [n_samples, n_features].

**Returns** Predicted values.

**Return type** pd.Series

**Raises** **MethodPropertyNotFoundError** – If estimator does not have a predict method or a component_obj that implements predict.

**predict_proba**(*self*, *X: pandas.DataFrame*) → pandas.Series

Make probability estimates for labels.

**Parameters** **X** (*pd.DataFrame*) – Features.

**Returns** Probability estimates.

**Return type** pd.Series

**Raises** **MethodPropertyNotFoundError** – If estimator does not have a predict_proba method or a component_obj that implements predict_proba.

**save**(*self*, *file_path*, *pickle_protocol=cloudpickle.DEFAULT_PROTOCOL*)

Saves component at file path.

**Parameters**

- **file_path** (*str*) – Location to save file.
- **pickle_protocol** (*int*) – The pickle data stream format.

**update_parameters**(*self*, *update_dict*, *reset_fit=True*)

Updates the parameter dictionary of the component.

**Parameters**

- **update_dict** (*dict*) – A dict of parameters to update.
- **reset_fit** (*bool, optional*) – If True, will set *_is_fitted* to False.

**class** evalml.pipelines.components.estimators.**XGBoostClassifier**(*eta=0.1*, *max_depth=6*, *min_child_weight=1*, *n_estimators=100*, *random_seed=0*, *eval_metric='logloss'*, *n_jobs=12*, ***kwargs*)

XGBoost Classifier.

**Parameters**

- **eta** (*float*) – Boosting learning rate. Defaults to 0.1.
- **max_depth** (*int*) – Maximum tree depth for base learners. Defaults to 6.
- **min_child_weight** (*float*) – Minimum sum of instance weight (hessian) needed in a child. Defaults to 1.0

- **n_estimators** (*int*) – Number of gradient boosted trees. Equivalent to number of boosting rounds. Defaults to 100.
- **random_seed** (*int*) – Seed for the random number generator. Defaults to 0.
- **n_jobs** (*int*) – Number of parallel threads used to run xgboost. Note that creating thread contention will significantly slow down the algorithm. Defaults to 12.

**Attributes**

| | |
|---|---|
| **hyper-parameter_ranges** | { "eta": Real(0.000001, 1), "max_depth": Integer(1, 10), "min_child_weight": Real(1, 10), "n_estimators": Integer(1, 1000),} |
| **model_family** | ModelFamily.XGBOOST |
| **modifies_features** | True |
| **modifies_target** | False |
| **name** | XGBoost Classifier |
| **SEED_MAX** | None |
| **SEED_MIN** | None |
| **supported_problem_types** | [ ProblemTypes.BINARY, ProblemTypes.MULTICLASS, ProblemTypes.TIME_SERIES_BINARY, ProblemTypes.TIME_SERIES_MULTICLASS,] |
| **training_only** | False |

**Methods**

| | |
|---|---|
| *clone* | Constructs a new component with the same parameters and random state. |
| *default_parameters* | Returns the default parameters for this component. |
| *describe* | Describe a component and its parameters. |
| *feature_importance* | Feature importance of fitted XGBoost classifier. |
| *fit* | Fits XGBoost classifier component to data. |
| *get_prediction_intervals* | Find the prediction intervals using the fitted regressor. |
| *load* | Loads component at file path. |
| *needs_fitting* | Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances. |
| *parameters* | Returns the parameters which were used to initialize the component. |
| *predict* | Make predictions using the fitted XGBoost classifier. |
| *predict_proba* | Make predictions using the fitted CatBoost classifier. |
| *save* | Saves component at file path. |
| *update_parameters* | Updates the parameter dictionary of the component. |

**clone**(*self*)

Constructs a new component with the same parameters and random state.

**Returns** A new instance of this component with identical parameters and random state.

**default_parameters**(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

**Returns** Default parameters for this component.

**Return type** dict

`describe`(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

**Parameters**

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description as dictionary in the format {"name": name, "parameters": parameters}

**Returns** Returns dictionary if return_dict is True, else None.

**Return type** None or dict

`property feature_importance`(*self*)

Feature importance of fitted XGBoost classifier.

`fit`(*self*, *X*, *y=None*)

Fits XGBoost classifier component to data.

**Parameters**

- **X** (*pd.DataFrame*) – The input training data of shape [n_samples, n_features].
- **y** (*pd.Series*) – The target training data of length [n_samples].

**Returns** self

`get_prediction_intervals`(*self*, *X: pandas.DataFrame*, *y: Optional[pandas.Series] = None*, *coverage: List[float] = None*, *predictions: pandas.Series = None*) → Dict[str, pandas.Series]

Find the prediction intervals using the fitted regressor.

This function takes the predictions of the fitted estimator and calculates the rolling standard deviation across all predictions using a window size of 5. The lower and upper predictions are determined by taking the percent point (quantile) function of the lower tail probability at each bound multiplied by the rolling standard deviation.

**Parameters**

- **X** (*pd.DataFrame*) – Data of shape [n_samples, n_features].
- **y** (*pd.Series*) – Target data. Ignored.
- **coverage** (*list[float]*) – A list of floats between the values 0 and 1 that the upper and lower bounds of the prediction interval should be calculated for.
- **predictions** (*pd.Series*) – Optional list of predictions to use. If None, will generate predictions using *X*.

**Returns** Prediction intervals, keys are in the format {coverage}_lower or {coverage}_upper.

**Return type** dict

**Raises** **MethodPropertyNotFoundError** – If the estimator does not support Time Series Regression as a problem type.

**static load**(*file_path*)

Loads component at file path.

**Parameters** **file_path** (*str*) – Location to load file.

**Returns** ComponentBase object

**needs_fitting**(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need to be fit or whose fit methods do nothing.

**Returns** True.

**property parameters**(*self*)

Returns the parameters which were used to initialize the component.

**predict**(*self*, *X*)

Make predictions using the fitted XGBoost classifier.

**Parameters** **X** (*pd.DataFrame*) – Data of shape [n_samples, n_features].

**Returns** Predicted values.

**Return type** pd.DataFrame

**predict_proba**(*self*, *X*)

Make predictions using the fitted CatBoost classifier.

**Parameters** **X** (*pd.DataFrame*) – Data of shape [n_samples, n_features].

**Returns** Predicted values.

**Return type** pd.DataFrame

**save**(*self*, *file_path*, *pickle_protocol=cloudpickle.DEFAULT_PROTOCOL*)

Saves component at file path.

**Parameters**

- **file_path** (*str*) – Location to save file.
- **pickle_protocol** (*int*) – The pickle data stream format.

**update_parameters**(*self*, *update_dict*, *reset_fit=True*)

Updates the parameter dictionary of the component.

**Parameters**

- **update_dict** (*dict*) – A dict of parameters to update.
- **reset_fit** (*bool, optional*) – If True, will set *_is_fitted* to False.

**class** evalml.pipelines.components.estimators.**XGBoostRegressor**(*eta: float = 0.1*, *max_depth: int = 6*, *min_child_weight: int = 1*, *n_estimators: int = 100*, *random_seed: Union[int, float] = 0*, *n_jobs: int = 12*, ***kwargs*)

XGBoost Regressor.

**Parameters**

- **eta** (*float*) – Boosting learning rate. Defaults to 0.1.

- **max_depth** (*int*) – Maximum tree depth for base learners. Defaults to 6.
- **min_child_weight** (*float*) – Minimum sum of instance weight (hessian) needed in a child. Defaults to 1.0
- **n_estimators** (*int*) – Number of gradient boosted trees. Equivalent to number of boosting rounds. Defaults to 100.
- **random_seed** (*int*) – Seed for the random number generator. Defaults to 0.
- **n_jobs** (*int*) – Number of parallel threads used to run xgboost. Note that creating thread contention will significantly slow down the algorithm. Defaults to 12.

**Attributes**

| | |
|---|---|
| **hyper-parameter_ranges** | { "eta": Real(0.000001, 1), "max_depth": Integer(1, 20), "min_child_weight": Real(1, 10), "n_estimators": Integer(1, 1000),} |
| **model_family** | ModelFamily.XGBOOST |
| **modifies_features** | True |
| **modifies_target** | False |
| **name** | XGBoost Regressor |
| **SEED_MAX** | None |
| **SEED_MIN** | None |
| **supported_problem_types** | [ ProblemTypes.REGRESSION, ProblemTypes.TIME_SERIES_REGRESSION,] |
| **training_only** | False |

**Methods**

| | |
|---|---|
| *clone* | Constructs a new component with the same parameters and random state. |
| *default_parameters* | Returns the default parameters for this component. |
| *describe* | Describe a component and its parameters. |
| *feature_importance* | Feature importance of fitted XGBoost regressor. |
| *fit* | Fits XGBoost regressor component to data. |
| *get_prediction_intervals* | Find the prediction intervals using the fitted XGBoostRegressor. |
| *load* | Loads component at file path. |
| *needs_fitting* | Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances. |
| *parameters* | Returns the parameters which were used to initialize the component. |
| *predict* | Make predictions using fitted XGBoost regressor. |
| *predict_proba* | Make probability estimates for labels. |
| *save* | Saves component at file path. |
| *update_parameters* | Updates the parameter dictionary of the component. |

**clone**(*self*)

Constructs a new component with the same parameters and random state.

**Returns** A new instance of this component with identical parameters and random state.

**default_parameters**(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

**Returns** Default parameters for this component.

**Return type** dict

**describe**(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

**Parameters**

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description as dictionary in the format {"name": name, "parameters": parameters}

**Returns** Returns dictionary if return_dict is True, else None.

**Return type** None or dict

**property feature_importance**(*self*) → pandas.Series

Feature importance of fitted XGBoost regressor.

**fit**(*self*, *X: pandas.DataFrame*, *y: Optional[pandas.Series] = None*)

Fits XGBoost regressor component to data.

**Parameters**

- **X** (*pd.DataFrame*) – The input training data of shape [n_samples, n_features].
- **y** (*pd.Series, optional*) – The target training data of length [n_samples].

**Returns** self

**get_prediction_intervals**(*self*, *X: pandas.DataFrame*, *y: Optional[pandas.Series] = None*, *coverage: List[float] = None*, *predictions: pandas.Series = None*) → Dict[str, pandas.Series]

Find the prediction intervals using the fitted XGBoostRegressor.

**Parameters**

- **X** (*pd.DataFrame*) – Data of shape [n_samples, n_features].
- **y** (*pd.Series*) – Target data. Ignored.
- **coverage** (*List[float]*) – A list of floats between the values 0 and 1 that the upper and lower bounds of the prediction interval should be calculated for.
- **predictions** (*pd.Series*) – Optional list of predictions to use. If None, will generate predictions using *X*.

**Returns** Prediction intervals, keys are in the format {coverage}_lower or {coverage}_upper.

**Return type** dict

**static load**(*file_path*)

Loads component at file path.

**Parameters** **file_path** (*str*) – Location to load file.

**Returns** ComponentBase object

**needs_fitting**(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need to be fit or whose fit methods do nothing.

**Returns** True.

**property parameters**(*self*)

Returns the parameters which were used to initialize the component.

**predict**(*self*, *X: pandas.DataFrame*) → pandas.Series

Make predictions using fitted XGBoost regressor.

**Parameters** **X** (*pd.DataFrame*) – Data of shape [n_samples, n_features].

**Returns** Predicted values.

**Return type** pd.Series

**predict_proba**(*self*, *X: pandas.DataFrame*) → pandas.Series

Make probability estimates for labels.

**Parameters** **X** (*pd.DataFrame*) – Features.

**Returns** Probability estimates.

**Return type** pd.Series

**Raises** **MethodPropertyNotFoundError** – If estimator does not have a predict_proba method or a component_obj that implements predict_proba.

**save**(*self*, *file_path*, *pickle_protocol=cloudpickle.DEFAULT_PROTOCOL*)

Saves component at file path.

**Parameters**

- **file_path** (*str*) – Location to save file.
- **pickle_protocol** (*int*) – The pickle data stream format.

**update_parameters**(*self*, *update_dict*, *reset_fit=True*)

Updates the parameter dictionary of the component.

**Parameters**

- **update_dict** (*dict*) – A dict of parameters to update.
- **reset_fit** (*bool, optional*) – If True, will set *_is_fitted* to False.

### transformers

Components that transform data.

### Subpackages

### dimensionality_reduction

Transformers that reduce the dimensionality of the input data.

### Submodules

### lda

Component that reduces the number of features by using Linear Discriminant Analysis.

### Module Contents

### Classes Summary

| | |
|---|---|
| *LinearDiscriminantAnalysis* | Reduces the number of features by using Linear Discriminant Analysis. |

### Contents

**class** evalml.pipelines.components.transformers.dimensionality_reduction.lda.**LinearDiscriminantAnalysis**(

Reduces the number of features by using Linear Discriminant Analysis.

**Parameters**

- **n_components** (*int*) – The number of features to maintain after computation. Defaults to None.
- **random_seed** (*int*) – Seed for the random number generator. Defaults to 0.

**Attributes**

| | |
|---|---|
| **hyper-parameter_ranges** | {} |
| **modifies_features** | True |
| **modifies_target** | False |
| **name** | Linear Discriminant Analysis Transformer |
| **training_only** | False |

**Methods**

| | |
|---|---|
| *clone* | Constructs a new component with the same parameters and random state. |
| *default_parameters* | Returns the default parameters for this component. |
| *describe* | Describe a component and its parameters. |
| *fit* | Fits the LDA component. |
| *fit_transform* | Fit and transform data using the LDA component. |
| *load* | Loads component at file path. |
| *needs_fitting* | Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances. |
| *parameters* | Returns the parameters which were used to initialize the component. |
| *save* | Saves component at file path. |
| *transform* | Transform data using the fitted LDA component. |
| *update_parameters* | Updates the parameter dictionary of the component. |

**clone**(*self*)

Constructs a new component with the same parameters and random state.

**Returns** A new instance of this component with identical parameters and random state.

**default_parameters**(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

**Returns** Default parameters for this component.

**Return type** dict

**describe**(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

**Parameters**

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description as dictionary in the format {"name": name, "parameters": parameters}

**Returns** Returns dictionary if return_dict is True, else None.

**Return type** None or dict

**fit**(*self*, *X*, *y*)

Fits the LDA component.

**Parameters**

- **X** (*pd.DataFrame*) – The input training data of shape [n_samples, n_features].
- **y** (*pd.Series, optional*) – The target training data of length [n_samples].

**Returns** self

**Raises** **ValueError** – If input data is not all numeric.

**fit_transform**(*self*, *X*, *y=None*)

Fit and transform data using the LDA component.

**Parameters**

- **X** (*pd.DataFrame*) – The input training data of shape [n_samples, n_features].
- **y** (*pd.Series, optional*) – The target training data of length [n_samples].

**Returns** Transformed data.

**Return type** pd.DataFrame

**Raises** **ValueError** – If input data is not all numeric.

**static load**(*file_path*)

Loads component at file path.

**Parameters** **file_path** (*str*) – Location to load file.

**Returns** ComponentBase object

**needs_fitting**(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need to be fit or whose fit methods do nothing.

**Returns** True.

**property parameters**(*self*)

Returns the parameters which were used to initialize the component.

**save**(*self*, *file_path*, *pickle_protocol=cloudpickle.DEFAULT_PROTOCOL*)

Saves component at file path.

**Parameters**

- **file_path** (*str*) – Location to save file.
- **pickle_protocol** (*int*) – The pickle data stream format.

**transform**(*self*, *X*, *y=None*)

Transform data using the fitted LDA component.

**Parameters**

- **X** (*pd.DataFrame*) – The input training data of shape [n_samples, n_features].
- **y** (*pd.Series, optional*) – The target training data of length [n_samples].

**Returns** Transformed data.

**Return type** pd.DataFrame

**Raises** **ValueError** – If input data is not all numeric.

**update_parameters**(*self*, *update_dict*, *reset_fit=True*)

Updates the parameter dictionary of the component.

**Parameters**

- **update_dict** (*dict*) – A dict of parameters to update.
- **reset_fit** (*bool, optional*) – If True, will set *_is_fitted* to False.

### pca

Component that reduces the number of features by using Principal Component Analysis (PCA).

### Module Contents

### Classes Summary

| | |
|---|---|
| *PCA* | Reduces the number of features by using Principal Component Analysis (PCA). |

### Contents

**class** evalml.pipelines.components.transformers.dimensionality_reduction.pca.**PCA**(*variance=0.95*, *n_components=None*, *random_seed=0*, ***kwargs*)

Reduces the number of features by using Principal Component Analysis (PCA).

**Parameters**

- **variance** (*float*) – The percentage of the original data variance that should be preserved when reducing the number of features. Defaults to 0.95.
- **n_components** (*int*) – The number of features to maintain after computing SVD. Defaults to None, but will override variance variable if set.
- **random_seed** (*int*) – Seed for the random number generator. Defaults to 0.

**Attributes**

| | |
|---|---|
| **hyperparameter_ranges** | Real(0.25, 1)}:type: {"variance" |
| **modifies_features** | True |
| **modifies_target** | False |
| **name** | PCA Transformer |
| **training_only** | False |

**Methods**

| | |
|---|---|
| *clone* | Constructs a new component with the same parameters and random state. |
| *default_parameters* | Returns the default parameters for this component. |
| *describe* | Describe a component and its parameters. |
| *fit* | Fits the PCA component. |
| *fit_transform* | Fit and transform data using the PCA component. |
| *load* | Loads component at file path. |
| *needs_fitting* | Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances. |
| *parameters* | Returns the parameters which were used to initialize the component. |
| *save* | Saves component at file path. |
| *transform* | Transform data using fitted PCA component. |
| *update_parameters* | Updates the parameter dictionary of the component. |

**clone**(*self*)

Constructs a new component with the same parameters and random state.

**Returns** A new instance of this component with identical parameters and random state.

**default_parameters**(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

**Returns** Default parameters for this component.

**Return type** dict

**describe**(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

**Parameters**

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description as dictionary in the format {"name": name, "parameters": parameters}

**Returns** Returns dictionary if return_dict is True, else None.

**Return type** None or dict

**fit**(*self*, *X*, *y=None*)

Fits the PCA component.

**Parameters**

- **X** (*pd.DataFrame*) – The input training data of shape [n_samples, n_features].
- **y** (*pd.Series, optional*) – The target training data of length [n_samples].

**Returns** self

**Raises** **ValueError** – If input data is not all numeric.

**fit_transform**(*self*, *X*, *y=None*)

Fit and transform data using the PCA component.

**Parameters**

- **X** (*pd.DataFrame*) – The input training data of shape [n_samples, n_features].
- **y** (*pd.Series, optional*) – The target training data of length [n_samples].

**Returns** Transformed data.

**Return type** pd.DataFrame

**Raises** **ValueError** – If input data is not all numeric.

**static load**(*file_path*)

Loads component at file path.

**Parameters** **file_path** (*str*) – Location to load file.

**Returns** ComponentBase object

**needs_fitting**(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need to be fit or whose fit methods do nothing.

**Returns** True.

**property parameters**(*self*)

Returns the parameters which were used to initialize the component.

**save**(*self*, *file_path*, *pickle_protocol=cloudpickle.DEFAULT_PROTOCOL*)

Saves component at file path.

**Parameters**

- **file_path** (*str*) – Location to save file.
- **pickle_protocol** (*int*) – The pickle data stream format.

**transform**(*self*, *X*, *y=None*)

Transform data using fitted PCA component.

**Parameters**

- **X** (*pd.DataFrame*) – The input training data of shape [n_samples, n_features].
- **y** (*pd.Series, optional*) – The target training data of length [n_samples].

**Returns** Transformed data.

**Return type** pd.DataFrame

**Raises** **ValueError** – If input data is not all numeric.

**update_parameters**(*self*, *update_dict*, *reset_fit=True*)

Updates the parameter dictionary of the component.

**Parameters**

- **update_dict** (*dict*) – A dict of parameters to update.
- **reset_fit** (*bool, optional*) – If True, will set *_is_fitted* to False.

### Package Contents

### Classes Summary

| | |
|---|---|
| *LinearDiscriminantAnalysis* | Reduces the number of features by using Linear Discriminant Analysis. |
| *PCA* | Reduces the number of features by using Principal Component Analysis (PCA). |

### Contents

**class** evalml.pipelines.components.transformers.dimensionality_reduction.**LinearDiscriminantAnalysis**(*n_com ran- dom_ **kwa*

Reduces the number of features by using Linear Discriminant Analysis.

**Parameters**

- **n_components** (*int*) – The number of features to maintain after computation. Defaults to None.
- **random_seed** (*int*) – Seed for the random number generator. Defaults to 0.

**Attributes**

| | |
|---|---|
| **hyper-parameter_ranges** | {} |
| **modifies_features** | True |
| **modifies_target** | False |
| **name** | Linear Discriminant Analysis Transformer |
| **training_only** | False |

**Methods**

| | |
|---|---|
| *clone* | Constructs a new component with the same parameters and random state. |
| *default_parameters* | Returns the default parameters for this component. |
| *describe* | Describe a component and its parameters. |
| *fit* | Fits the LDA component. |
| *fit_transform* | Fit and transform data using the LDA component. |
| *load* | Loads component at file path. |
| *needs_fitting* | Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances. |
| *parameters* | Returns the parameters which were used to initialize the component. |
| *save* | Saves component at file path. |
| *transform* | Transform data using the fitted LDA component. |
| *update_parameters* | Updates the parameter dictionary of the component. |

**clone**(*self*)

Constructs a new component with the same parameters and random state.

**Returns** A new instance of this component with identical parameters and random state.

**default_parameters**(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

**Returns** Default parameters for this component.

**Return type** dict

**describe**(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

**Parameters**

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description as dictionary in the format {"name": name, "parameters": parameters}

**Returns** Returns dictionary if return_dict is True, else None.

**Return type** None or dict

**fit**(*self*, *X*, *y*)

Fits the LDA component.

**Parameters**

- **X** (*pd.DataFrame*) – The input training data of shape [n_samples, n_features].
- **y** (*pd.Series, optional*) – The target training data of length [n_samples].

**Returns** self

**Raises** **ValueError** – If input data is not all numeric.

**fit_transform**(*self*, *X*, *y=None*)

Fit and transform data using the LDA component.

**Parameters**

- **X** (*pd.DataFrame*) – The input training data of shape [n_samples, n_features].
- **y** (*pd.Series, optional*) – The target training data of length [n_samples].

**Returns** Transformed data.

**Return type** pd.DataFrame

**Raises** **ValueError** – If input data is not all numeric.

**static load**(*file_path*)

Loads component at file path.

**Parameters** **file_path** (*str*) – Location to load file.

**Returns** ComponentBase object

**needs_fitting**(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need to be fit or whose fit methods do nothing.

**Returns** True.

**property parameters**(*self*)

Returns the parameters which were used to initialize the component.

**save**(*self*, *file_path*, *pickle_protocol=cloudpickle.DEFAULT_PROTOCOL*)

Saves component at file path.

**Parameters**

- **file_path** (*str*) – Location to save file.
- **pickle_protocol** (*int*) – The pickle data stream format.

**transform**(*self*, *X*, *y=None*)

Transform data using the fitted LDA component.

**Parameters**

- **X** (*pd.DataFrame*) – The input training data of shape [n_samples, n_features].
- **y** (*pd.Series, optional*) – The target training data of length [n_samples].

**Returns** Transformed data.

**Return type** pd.DataFrame

**Raises** **ValueError** – If input data is not all numeric.

**update_parameters**(*self*, *update_dict*, *reset_fit=True*)

Updates the parameter dictionary of the component.

**Parameters**

- **update_dict** (*dict*) – A dict of parameters to update.
- **reset_fit** (*bool, optional*) – If True, will set *_is_fitted* to False.

**class** evalml.pipelines.components.transformers.dimensionality_reduction.**PCA**(*variance=0.95*, *n_components=None*, *random_seed=0*, ***kwargs*)

Reduces the number of features by using Principal Component Analysis (PCA).

**Parameters**

- **variance** (*float*) – The percentage of the original data variance that should be preserved when reducing the number of features. Defaults to 0.95.
- **n_components** (*int*) – The number of features to maintain after computing SVD. Defaults to None, but will override variance variable if set.
- **random_seed** (*int*) – Seed for the random number generator. Defaults to 0.

**Attributes**

| | |
|---|---|
| **hyper-parameter_ranges** | Real(0.25, 1)}:type: {"variance" |
| **modifies_features** | True |
| **modifies_target** | False |
| **name** | PCA Transformer |
| **training_only** | False |

**Methods**

| | |
|---|---|
| *clone* | Constructs a new component with the same parameters and random state. |
| *default_parameters* | Returns the default parameters for this component. |
| *describe* | Describe a component and its parameters. |
| *fit* | Fits the PCA component. |
| *fit_transform* | Fit and transform data using the PCA component. |
| *load* | Loads component at file path. |
| *needs_fitting* | Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances. |
| *parameters* | Returns the parameters which were used to initialize the component. |
| *save* | Saves component at file path. |
| *transform* | Transform data using fitted PCA component. |
| *update_parameters* | Updates the parameter dictionary of the component. |

**clone**(*self*)

Constructs a new component with the same parameters and random state.

**Returns** A new instance of this component with identical parameters and random state.

**default_parameters**(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

**Returns** Default parameters for this component.

**Return type** dict

**describe**(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

**Parameters**

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description as dictionary in the format {"name": name, "parameters": parameters}

**Returns** Returns dictionary if return_dict is True, else None.

**Return type** None or dict

**fit**(*self*, *X*, *y=None*)

Fits the PCA component.

**Parameters**

- **X** (*pd.DataFrame*) – The input training data of shape [n_samples, n_features].
- **y** (*pd.Series, optional*) – The target training data of length [n_samples].

**Returns** self

**Raises** **ValueError** – If input data is not all numeric.

**fit_transform**(*self*, *X*, *y=None*)

Fit and transform data using the PCA component.

**Parameters**

- **X** (*pd.DataFrame*) – The input training data of shape [n_samples, n_features].
- **y** (*pd.Series, optional*) – The target training data of length [n_samples].

**Returns** Transformed data.

**Return type** pd.DataFrame

**Raises** **ValueError** – If input data is not all numeric.

**static load**(*file_path*)

Loads component at file path.

**Parameters** **file_path** (*str*) – Location to load file.

**Returns** ComponentBase object

**needs_fitting**(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need to be fit or whose fit methods do nothing.

**Returns** True.

**property parameters**(*self*)

Returns the parameters which were used to initialize the component.

**save**(*self*, *file_path*, *pickle_protocol=cloudpickle.DEFAULT_PROTOCOL*)

Saves component at file path.

**Parameters**

- **file_path** (*str*) – Location to save file.

- **pickle_protocol** (*int*) – The pickle data stream format.

**transform**(*self*, *X*, *y=None*)

Transform data using fitted PCA component.

**Parameters**

- **X** (*pd.DataFrame*) – The input training data of shape [n_samples, n_features].
- **y** (*pd.Series, optional*) – The target training data of length [n_samples].

**Returns** Transformed data.

**Return type** pd.DataFrame

**Raises** **ValueError** – If input data is not all numeric.

**update_parameters**(*self*, *update_dict*, *reset_fit=True*)

Updates the parameter dictionary of the component.

**Parameters**

- **update_dict** (*dict*) – A dict of parameters to update.
- **reset_fit** (*bool, optional*) – If True, will set *_is_fitted* to False.

## encoders

Components used to encode the input data.

### Submodules

### label_encoder

A transformer that encodes target labels using values between 0 and num_classes - 1.

### Module Contents

### Classes Summary

| | |
|---|---|
| *LabelEncoder* | A transformer that encodes target labels using values between 0 and num_classes - 1. |

### Contents

**class** evalml.pipelines.components.transformers.encoders.label_encoder.**LabelEncoder**(*positive_label=None*, *random_seed=0*, ***kwargs*)

A transformer that encodes target labels using values between 0 and num_classes - 1.

**Parameters**

- **positive_label** (*int, str*) – The label for the class that should be treated as positive (1) for binary classification problems. Ignored for multiclass problems. Defaults to None.
- **random_seed** (*int*) – Seed for the random number generator. Defaults to 0. Ignored.

**Attributes**

| | |
|---|---|
| **hyper-parameter_ranges** | {} |
| **modifies_features** | False |
| **modifies_target** | True |
| **name** | Label Encoder |
| **training_only** | False |

**Methods**

| | |
|---|---|
| *clone* | Constructs a new component with the same parameters and random state. |
| *default_parameters* | Returns the default parameters for this component. |
| *describe* | Describe a component and its parameters. |
| *fit* | Fits the label encoder. |
| *fit_transform* | Fit and transform data using the label encoder. |
| *inverse_transform* | Decodes the target data. |
| *load* | Loads component at file path. |
| *needs_fitting* | Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances. |
| *parameters* | Returns the parameters which were used to initialize the component. |
| *save* | Saves component at file path. |
| *transform* | Transform the target using the fitted label encoder. |
| *update_parameters* | Updates the parameter dictionary of the component. |

**clone**(*self*)

Constructs a new component with the same parameters and random state.

**Returns** A new instance of this component with identical parameters and random state.

**default_parameters**(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

**Returns** Default parameters for this component.

**Return type** dict

**describe**(*self, print_name=False, return_dict=False*)

Describe a component and its parameters.

**Parameters**

- **print_name** (*bool, optional*) – whether to print name of component

- **return_dict** (*bool, optional*) – whether to return description as dictionary in the format {"name": name, "parameters": parameters}

**Returns** Returns dictionary if return_dict is True, else None.

**Return type** None or dict

**fit**(*self*, *X*, *y*)

Fits the label encoder.

**Parameters**

- **X** (*pd.DataFrame*) – The input training data of shape [n_samples, n_features]. Ignored.
- **y** (*pd.Series*) – The target training data of length [n_samples].

**Returns** self

**Raises** **ValueError** – If input *y* is None.

**fit_transform**(*self*, *X*, *y*)

Fit and transform data using the label encoder.

**Parameters**

- **X** (*pd.DataFrame*) – The input training data of shape [n_samples, n_features].
- **y** (*pd.Series*) – The target training data of length [n_samples].

**Returns** The original features and an encoded version of the target.

**Return type** pd.DataFrame, pd.Series

**inverse_transform**(*self*, *y*)

Decodes the target data.

**Parameters** **y** (*pd.Series*) – Target data.

**Returns** The decoded version of the target.

**Return type** pd.Series

**Raises** **ValueError** – If input *y* is None.

**static load**(*file_path*)

Loads component at file path.

**Parameters** **file_path** (*str*) – Location to load file.

**Returns** ComponentBase object

**needs_fitting**(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need to be fit or whose fit methods do nothing.

**Returns** True.

**property parameters**(*self*)

Returns the parameters which were used to initialize the component.

**save**(*self*, *file_path*, *pickle_protocol=cloudpickle.DEFAULT_PROTOCOL*)

Saves component at file path.

**Parameters**

- **file_path** (*str*) – Location to save file.
- **pickle_protocol** (*int*) – The pickle data stream format.

**transform**(*self*, *X*, *y=None*)

Transform the target using the fitted label encoder.

**Parameters**

- **X** (*pd.DataFrame*) – The input training data of shape [n_samples, n_features]. Ignored.
- **y** (*pd.Series*) – The target training data of length [n_samples].

**Returns** The original features and an encoded version of the target.

**Return type** pd.DataFrame, pd.Series

**Raises** **ValueError** – If input *y* is None.

**update_parameters**(*self*, *update_dict*, *reset_fit=True*)

Updates the parameter dictionary of the component.

**Parameters**

- **update_dict** (*dict*) – A dict of parameters to update.
- **reset_fit** (*bool, optional*) – If True, will set *_is_fitted* to False.

### onehot_encoder

A transformer that encodes categorical features in a one-hot numeric array.

### Module Contents

### Classes Summary

| | |
|---|---|
| *OneHotEncoder* | A transformer that encodes categorical features in a one-hot numeric array. |
| *OneHotEncoderMeta* | A version of the ComponentBaseMeta class which includes validation on an additional one-hot-encoder-specific method *categories*. |

### Contents

**class** evalml.pipelines.components.transformers.encoders.onehot_encoder.**OneHotEncoder**(*top_n=10, features_to_encode=None, categories=None, drop='if_binary', handle_unknown='ignore', handle_missing='error', random_seed=0, **kwargs*)

A transformer that encodes categorical features in a one-hot numeric array.

**Parameters**

- **top_n** (*int*) – Number of categories per column to encode. If None, all categories will be encoded. Otherwise, the *n* most frequent will be encoded and all others will be dropped. Defaults to 10.
- **features_to_encode** (*list[str]*) – List of columns to encode. All other columns will remain untouched. If None, all appropriate columns will be encoded. Defaults to None.
- **categories** (*list*) – A two dimensional list of categories, where *categories[i]* is a list of the categories for the column at index *i*. This can also be *None*, or *"auto"* if *top_n* is not None. Defaults to None.
- **drop** (*string, list*) – Method ("first" or "if_binary") to use to drop one category per feature. Can also be a list specifying which categories to drop for each feature. Defaults to 'if_binary'.
- **handle_unknown** (*string*) – Whether to ignore or error for unknown categories for a feature encountered during *fit* or *transform*. If either *top_n* or *categories* is used to limit the number of categories per column, this must be "ignore". Defaults to "ignore".
- **handle_missing** (*string*) – Options for how to handle missing (NaN) values encountered during *fit* or *transform*. If this is set to "as_category" and NaN values are within the *n* most frequent, "nan" values will be encoded as their own column. If this is set to "error", any missing values encountered will raise an error. Defaults to "error".
- **random_seed** (*int*) – Seed for the random number generator. Defaults to 0.

**Attributes**

| | |
|---|---|
| **hyperparameter_ranges** | {} |
| **modifies_features** | True |
| **modifies_target** | False |
| **name** | One Hot Encoder |
| **training_only** | False |

**Methods**

| | |
|---|---|
| *categories* | Returns a list of the unique categories to be encoded for the particular feature, in order. |
| *clone* | Constructs a new component with the same parameters and random state. |
| *default_parameters* | Returns the default parameters for this component. |
| *describe* | Describe a component and its parameters. |
| *fit* | Fits the one-hot encoder component. |
| *fit_transform* | Fits on X and transforms X. |
| *get_feature_names* | Return feature names for the categorical features after fitting. |
| *load* | Loads component at file path. |
| *needs_fitting* | Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances. |
| *parameters* | Returns the parameters which were used to initialize the component. |
| *save* | Saves component at file path. |
| *transform* | One-hot encode the input data. |
| *update_parameters* | Updates the parameter dictionary of the component. |

**categories**(*self*, *feature_name*)

Returns a list of the unique categories to be encoded for the particular feature, in order.

**Parameters** **feature_name** (*str*) – The name of any feature provided to one-hot encoder during fit.

**Returns** The unique categories, in the same dtype as they were provided during fit.

**Return type** np.ndarray

**Raises** **ValueError** – If feature was not provided to one-hot encoder as a training feature.

**clone**(*self*)

Constructs a new component with the same parameters and random state.

**Returns** A new instance of this component with identical parameters and random state.

**default_parameters**(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

**Returns** Default parameters for this component.

**Return type** dict

**describe**(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

**Parameters**

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description as dictionary in the format {"name": name, "parameters": parameters}

**Returns** Returns dictionary if return_dict is True, else None.

**Return type** None or dict

**fit**(*self*, *X*, *y=None*)

Fits the one-hot encoder component.

**Parameters**

- **X** (*pd.DataFrame*) – The input training data of shape [n_samples, n_features].
- **y** (*pd.Series, optional*) – The target training data of length [n_samples].

**Returns** self

**Raises** **ValueError** – If encoding a column failed.

**fit_transform**(*self*, *X*, *y=None*)

Fits on X and transforms X.

**Parameters**

- **X** (*pd.DataFrame*) – Data to fit and transform.
- **y** (*pd.Series*) – Target data.

**Returns** Transformed X.

**Return type** pd.DataFrame

**Raises** **MethodPropertyNotFoundError** – If transformer does not have a transform method or a component_obj that implements transform.

**get_feature_names**(*self*)

Return feature names for the categorical features after fitting.

Feature names are formatted as {column name}_{category name}. In the event of a duplicate name, an integer will be added at the end of the feature name to distinguish it.

For example, consider a dataframe with a column called "A" and category "x_y" and another column called "A_x" with "y". In this example, the feature names would be "A_x_y" and "A_x_y_1".

**Returns** The feature names after encoding, provided in the same order as input_features.

**Return type** np.ndarray

**static load**(*file_path*)

Loads component at file path.

**Parameters** **file_path** (*str*) – Location to load file.

**Returns** ComponentBase object

**needs_fitting**(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need to be fit or whose fit methods do nothing.

**Returns** True.

**property parameters**(*self*)

Returns the parameters which were used to initialize the component.

**save**(*self*, *file_path*, *pickle_protocol=cloudpickle.DEFAULT_PROTOCOL*)

Saves component at file path.

**Parameters**

- **file_path** (*str*) – Location to save file.

- **pickle_protocol** (*int*) – The pickle data stream format.

**transform**(*self*, *X*, *y=None*)

One-hot encode the input data.

**Parameters**

- **X** (*pd.DataFrame*) – Features to one-hot encode.
- **y** (*pd.Series*) – Ignored.

**Returns** Transformed data, where each categorical feature has been encoded into numerical columns using one-hot encoding.

**Return type** pd.DataFrame

**update_parameters**(*self*, *update_dict*, *reset_fit=True*)

Updates the parameter dictionary of the component.

**Parameters**

- **update_dict** (*dict*) – A dict of parameters to update.
- **reset_fit** (*bool, optional*) – If True, will set *_is_fitted* to False.

**class** evalml.pipelines.components.transformers.encoders.onehot_encoder.**OneHotEncoderMeta**

A version of the ComponentBaseMeta class which includes validation on an additional one-hot-encoder-specific method *categories*.

**Attributes**

| | |
|---|---|
| **FIT_METHODS** | ['fit', 'fit_transform'] |
| **METHODS_TO_CHECK** | None |
| **PROPERTIES_TO_CHECK** | ['feature_importance'] |

**Methods**

| | |
|---|---|
| *check_for_fit* | *check_for_fit* wraps a method that validates if *self._is_fitted* is *True*. |
| *register* | Register a virtual subclass of an ABC. |
| *set_fit* | Wrapper for the fit method. |

**classmethod check_for_fit**(*cls*, *method*)

*check_for_fit* wraps a method that validates if *self._is_fitted* is *True*.

It raises an exception if *False* and calls and returns the wrapped method if *True*.

**Parameters** **method** (*callable*) – Method to wrap.

**Returns** The wrapped method.

**Raises** **ComponentNotYetFittedError** – If component is not yet fitted.

**register**(*cls*, *subclass*)

Register a virtual subclass of an ABC.

Returns the subclass, to allow usage as a class decorator.

**classmethod set_fit**(*cls*, *method*)

Wrapper for the fit method.

### ordinal_encoder

A transformer that encodes ordinal features as an array of ordinal integers representing the relative order of categories.

### Module Contents

### Classes Summary

| | |
|---|---|
| *OrdinalEncoder* | A transformer that encodes ordinal features as an array of ordinal integers representing the relative order of categories. |
| *OrdinalEncoderMeta* | A version of the ComponentBaseMeta class which includes validation on an additional ordinal-encoder-specific method *categories*. |

### Contents

**class** evalml.pipelines.components.transformers.encoders.ordinal_encoder.**OrdinalEncoder**(*features_to_encode=N cat-e-gories=None, han-dle_unknown='error' un-known_value=None, en-coded_missing_value ran-dom_seed=0, **kwargs*)

A transformer that encodes ordinal features as an array of ordinal integers representing the relative order of categories.

**Parameters**

- **features_to_encode** (*list[str]*) – List of columns to encode. All other columns will remain untouched. If None, all appropriate columns will be encoded. Defaults to None. The order of columns does not matter.
- **categories** (*dict[str, list[str]]*) – A dictionary mapping column names to their categories in the dataframes passed in at fit and transform. The order of categories specified for a column does not matter. Any category found in the data that is not present in categories will be handled as an unknown value. To not have unknown values raise an error, set handle_unknown to "use_encoded_value". Defaults to None.
- **handle_unknown** (*"error" or "use_encoded_value"*) – Whether to ignore or error for unknown categories for a feature encountered during *fit* or *transform*. When set to "error", an error will be raised when an unknown category is found. When set to "use_encoded_value",

unknown categories will be encoded as the value given for the parameter unknown_value. Defaults to "error."

- **unknown_value** (*int or np.nan*) – The value to use for unknown categories seen during fit or transform. Required when the parameter handle_unknown is set to "use_encoded_value." The value has to be distinct from the values used to encode any of the categories in fit. Defaults to None.
- **encoded_missing_value** (*int or np.nan*) – The value to use for missing (null) values seen during fit or transform. Defaults to np.nan.
- **random_seed** (*int*) – Seed for the random number generator. Defaults to 0.

**Attributes**

| | |
|---|---|
| **hyperparameter_ranges** | {} |
| **modifies_features** | True |
| **modifies_target** | False |
| **name** | Ordinal Encoder |
| **training_only** | False |

**Methods**

| | |
|---|---|
| *categories* | Returns a list of the unique categories to be encoded for the particular feature, in order. |
| *clone* | Constructs a new component with the same parameters and random state. |
| *default_parameters* | Returns the default parameters for this component. |
| *describe* | Describe a component and its parameters. |
| *fit* | Fits the ordinal encoder component. |
| *fit_transform* | Fits on X and transforms X. |
| *get_feature_names* | Return feature names for the ordinal features after fitting. |
| *load* | Loads component at file path. |
| *needs_fitting* | Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances. |
| *parameters* | Returns the parameters which were used to initialize the component. |
| *save* | Saves component at file path. |
| *transform* | Ordinally encode the input data. |
| *update_parameters* | Updates the parameter dictionary of the component. |

**categories**(*self*, *feature_name*)

Returns a list of the unique categories to be encoded for the particular feature, in order.

**Parameters** **feature_name** (*str*) – The name of any feature provided to ordinal encoder during fit.

**Returns** The unique categories, in the same dtype as they were provided during fit.

**Return type** np.ndarray

**Raises** `ValueError` – If feature was not provided to ordinal encoder as a training feature.

`clone`(*self*)

Constructs a new component with the same parameters and random state.

**Returns** A new instance of this component with identical parameters and random state.

`default_parameters`(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

**Returns** Default parameters for this component.

**Return type** dict

`describe`(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

**Parameters**

- `print_name` (*bool, optional*) – whether to print name of component
- `return_dict` (*bool, optional*) – whether to return description as dictionary in the format {"name": name, "parameters": parameters}

**Returns** Returns dictionary if return_dict is True, else None.

**Return type** None or dict

`fit`(*self*, *X*, *y=None*)

Fits the ordinal encoder component.

**Parameters**

- `X` (*pd.DataFrame*) – The input training data of shape [n_samples, n_features].
- `y` (*pd.Series, optional*) – The target training data of length [n_samples].

**Returns** self

**Raises**

- `ValueError` – If encoding a column failed.
- `TypeError` – If non-Ordinal columns are specified in features_to_encode.

`fit_transform`(*self*, *X*, *y=None*)

Fits on X and transforms X.

**Parameters**

- `X` (*pd.DataFrame*) – Data to fit and transform.
- `y` (*pd.Series*) – Target data.

**Returns** Transformed X.

**Return type** pd.DataFrame

**Raises** `MethodPropertyNotFoundError` – If transformer does not have a transform method or a component_obj that implements transform.

**get_feature_names**(*self*)

Return feature names for the ordinal features after fitting.

Feature names are formatted as {column name}_ordinal_encoding.

**Returns** The feature names after encoding, provided in the same order as input_features.

**Return type** np.ndarray

**static load**(*file_path*)

Loads component at file path.

**Parameters** **file_path** (*str*) – Location to load file.

**Returns** ComponentBase object

**needs_fitting**(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need to be fit or whose fit methods do nothing.

**Returns** True.

**property parameters**(*self*)

Returns the parameters which were used to initialize the component.

**save**(*self*, *file_path*, *pickle_protocol=cloudpickle.DEFAULT_PROTOCOL*)

Saves component at file path.

**Parameters**

- **file_path** (*str*) – Location to save file.
- **pickle_protocol** (*int*) – The pickle data stream format.

**transform**(*self*, *X*, *y=None*)

Ordinally encode the input data.

**Parameters**

- **X** (*pd.DataFrame*) – Features to encode.
- **y** (*pd.Series*) – Ignored.

**Returns** Transformed data, where each ordinal feature has been encoded into a numerical column where ordinal integers represent the relative order of categories.

**Return type** pd.DataFrame

**update_parameters**(*self*, *update_dict*, *reset_fit=True*)

Updates the parameter dictionary of the component.

**Parameters**

- **update_dict** (*dict*) – A dict of parameters to update.
- **reset_fit** (*bool, optional*) – If True, will set *_is_fitted* to False.

**class** evalml.pipelines.components.transformers.encoders.ordinal_encoder.**OrdinalEncoderMeta**

A version of the ComponentBaseMeta class which includes validation on an additional ordinal-encoder-specific method *categories*.

**Attributes**

| | |
|---|---|
| **FIT_METHODS** | ['fit', 'fit_transform'] |
| **METHODS_TO_CHECK** | None |
| **PROPERTIES_TO_CHECK** | ['feature_importance'] |

**Methods**

| | |
|---|---|
| *check_for_fit* | *check_for_fit* wraps a method that validates if *self._is_fitted* is *True*. |
| *register* | Register a virtual subclass of an ABC. |
| *set_fit* | Wrapper for the fit method. |

**classmethod check_for_fit**(*cls*, *method*)

*check_for_fit* wraps a method that validates if *self._is_fitted* is *True*.

It raises an exception if *False* and calls and returns the wrapped method if *True*.

**Parameters** **method** (*callable*) – Method to wrap.

**Returns** The wrapped method.

**Raises** **ComponentNotYetFittedError** – If component is not yet fitted.

**register**(*cls*, *subclass*)

Register a virtual subclass of an ABC.

Returns the subclass, to allow usage as a class decorator.

**classmethod set_fit**(*cls*, *method*)

Wrapper for the fit method.

## target_encoder

A transformer that encodes categorical features into target encodings.

### Module Contents

### Classes Summary

| | |
|---|---|
| *TargetEncoder* | A transformer that encodes categorical features into target encodings. |

### Contents

**class** evalml.pipelines.components.transformers.encoders.target_encoder.**TargetEncoder**(*cols=None, smoothing=1, handle_unknown='value', handle_missing='value', random_seed=0, **kwargs*)

A transformer that encodes categorical features into target encodings.

**Parameters**

- **cols** (*list*) – Columns to encode. If None, all string columns will be encoded, otherwise only the columns provided will be encoded. Defaults to None
- **smoothing** (*float*) – The smoothing factor to apply. The larger this value is, the more influence the expected target value has on the resulting target encodings. Must be strictly larger than 0. Defaults to 1.0
- **handle_unknown** (*string*) – Determines how to handle unknown categories for a feature encountered. Options are 'value', 'error', nd 'return_nan'. Defaults to 'value', which replaces with the target mean
- **handle_missing** (*string*) – Determines how to handle missing values encountered during *fit* or *transform*. Options are 'value', 'error', and 'return_nan'. Defaults to 'value', which replaces with the target mean
- **random_seed** (*int*) – Seed for the random number generator. Defaults to 0.

**Attributes**

| | |
|---|---|
| **hyperparameter_ranges** | {} |
| **modifies_features** | True |
| **modifies_target** | False |
| **name** | Target Encoder |
| **training_only** | False |

**Methods**

| | |
|---|---|
| *clone* | Constructs a new component with the same parameters and random state. |
| *default_parameters* | Returns the default parameters for this component. |
| *describe* | Describe a component and its parameters. |
| *fit* | Fits the target encoder. |
| *fit_transform* | Fit and transform data using the target encoder. |
| *get_feature_names* | Return feature names for the input features after fitting. |
| *load* | Loads component at file path. |
| *needs_fitting* | Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances. |
| *parameters* | Returns the parameters which were used to initialize the component. |
| *save* | Saves component at file path. |
| *transform* | Transform data using the fitted target encoder. |
| *update_parameters* | Updates the parameter dictionary of the component. |

**clone**(*self*)

Constructs a new component with the same parameters and random state.

**Returns** A new instance of this component with identical parameters and random state.

**default_parameters**(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

**Returns** Default parameters for this component.

**Return type** dict

**describe**(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

**Parameters**

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description as dictionary in the format {"name": name, "parameters": parameters}

**Returns** Returns dictionary if return_dict is True, else None.

**Return type** None or dict

**fit**(*self*, *X*, *y*)

Fits the target encoder.

**Parameters**

- **X** (*pd.DataFrame*) – The input training data of shape [n_samples, n_features].
- **y** (*pd.Series, optional*) – The target training data of length [n_samples].

**Returns** self

**fit_transform**(*self*, *X*, *y*)

Fit and transform data using the target encoder.

**Parameters**

- **X** (*pd.DataFrame*) – The input training data of shape [n_samples, n_features].
- **y** (*pd.Series, optional*) – The target training data of length [n_samples].

**Returns** Transformed data.

**Return type** pd.DataFrame

**get_feature_names**(*self*)

Return feature names for the input features after fitting.

**Returns** The feature names after encoding.

**Return type** np.array

**static load**(*file_path*)

Loads component at file path.

**Parameters** **file_path** (*str*) – Location to load file.

**Returns** ComponentBase object

**needs_fitting**(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need to be fit or whose fit methods do nothing.

**Returns** True.

**property parameters**(*self*)

Returns the parameters which were used to initialize the component.

**save**(*self*, *file_path*, *pickle_protocol=cloudpickle.DEFAULT_PROTOCOL*)

Saves component at file path.

**Parameters**

- **file_path** (*str*) – Location to save file.
- **pickle_protocol** (*int*) – The pickle data stream format.

**transform**(*self*, *X*, *y=None*)

Transform data using the fitted target encoder.

**Parameters**

- **X** (*pd.DataFrame*) – The input training data of shape [n_samples, n_features].
- **y** (*pd.Series, optional*) – The target training data of length [n_samples].

**Returns** Transformed data.

**Return type** pd.DataFrame

**update_parameters**(*self*, *update_dict*, *reset_fit=True*)

Updates the parameter dictionary of the component.

**Parameters**

- **update_dict** (*dict*) – A dict of parameters to update.
- **reset_fit** (*bool, optional*) – If True, will set *_is_fitted* to False.

## Package Contents

## Classes Summary

| | |
|---|---|
| *LabelEncoder* | A transformer that encodes target labels using values between 0 and num_classes - 1. |
| *OneHotEncoder* | A transformer that encodes categorical features in a one-hot numeric array. |
| *OrdinalEncoder* | A transformer that encodes ordinal features as an array of ordinal integers representing the relative order of categories. |
| *TargetEncoder* | A transformer that encodes categorical features into target encodings. |

## Contents

**class** evalml.pipelines.components.transformers.encoders.**LabelEncoder**(*positive_label=None*, *random_seed=0*, ***kwargs*)

A transformer that encodes target labels using values between 0 and num_classes - 1.

**Parameters**

- **positive_label** (*int, str*) – The label for the class that should be treated as positive (1) for binary classification problems. Ignored for multiclass problems. Defaults to None.
- **random_seed** (*int*) – Seed for the random number generator. Defaults to 0. Ignored.

**Attributes**

| | |
|---|---|
| **hyper-parameter_ranges** | {} |
| **modifies_features** | False |
| **modifies_target** | True |
| **name** | Label Encoder |
| **training_only** | False |

**Methods**

| | |
|---|---|
| *clone* | Constructs a new component with the same parameters and random state. |
| *default_parameters* | Returns the default parameters for this component. |
| *describe* | Describe a component and its parameters. |
| *fit* | Fits the label encoder. |
| *fit_transform* | Fit and transform data using the label encoder. |
| *inverse_transform* | Decodes the target data. |
| *load* | Loads component at file path. |
| *needs_fitting* | Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances. |
| *parameters* | Returns the parameters which were used to initialize the component. |
| *save* | Saves component at file path. |
| *transform* | Transform the target using the fitted label encoder. |
| *update_parameters* | Updates the parameter dictionary of the component. |

**clone**(*self*)

Constructs a new component with the same parameters and random state.

**Returns** A new instance of this component with identical parameters and random state.

**default_parameters**(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

**Returns** Default parameters for this component.

**Return type** dict

**describe**(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

**Parameters**

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description as dictionary in the format {"name": name, "parameters": parameters}

**Returns** Returns dictionary if return_dict is True, else None.

**Return type** None or dict

**fit**(*self*, *X*, *y*)

Fits the label encoder.

**Parameters**

- **X** (*pd.DataFrame*) – The input training data of shape [n_samples, n_features]. Ignored.
- **y** (*pd.Series*) – The target training data of length [n_samples].

**Returns** self

**Raises** **ValueError** – If input *y* is None.

**fit_transform**(*self*, *X*, *y*)

Fit and transform data using the label encoder.

**Parameters**

- **X** (*pd.DataFrame*) – The input training data of shape [n_samples, n_features].
- **y** (*pd.Series*) – The target training data of length [n_samples].

**Returns** The original features and an encoded version of the target.

**Return type** pd.DataFrame, pd.Series

**inverse_transform**(*self*, *y*)

Decodes the target data.

**Parameters** **y** (*pd.Series*) – Target data.

**Returns** The decoded version of the target.

**Return type** pd.Series

**Raises** **ValueError** – If input *y* is None.

**static load**(*file_path*)

Loads component at file path.

**Parameters** **file_path** (*str*) – Location to load file.

**Returns** ComponentBase object

**needs_fitting**(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need to be fit or whose fit methods do nothing.

**Returns** True.

**property parameters**(*self*)

Returns the parameters which were used to initialize the component.

**save**(*self*, *file_path*, *pickle_protocol=cloudpickle.DEFAULT_PROTOCOL*)

Saves component at file path.

**Parameters**

- **file_path** (*str*) – Location to save file.
- **pickle_protocol** (*int*) – The pickle data stream format.

**transform**(*self*, *X*, *y=None*)

Transform the target using the fitted label encoder.

**Parameters**

- **X** (*pd.DataFrame*) – The input training data of shape [n_samples, n_features]. Ignored.
- **y** (*pd.Series*) – The target training data of length [n_samples].

**Returns** The original features and an encoded version of the target.

**Return type** pd.DataFrame, pd.Series

**Raises** **ValueError** – If input *y* is None.

**update_parameters**(*self*, *update_dict*, *reset_fit=True*)

Updates the parameter dictionary of the component.

**Parameters**

- **update_dict** (*dict*) – A dict of parameters to update.
- **reset_fit** (*bool, optional*) – If True, will set *_is_fitted* to False.

**class** evalml.pipelines.components.transformers.encoders.**OneHotEncoder**(*top_n=10*, *features_to_encode=None*, *categories=None*, *drop='if_binary'*, *handle_unknown='ignore'*, *handle_missing='error'*, *random_seed=0*, ***kwargs*)

A transformer that encodes categorical features in a one-hot numeric array.

**Parameters**

- **top_n** (*int*) – Number of categories per column to encode. If None, all categories will be encoded. Otherwise, the *n* most frequent will be encoded and all others will be dropped. Defaults to 10.
- **features_to_encode** (*list[str]*) – List of columns to encode. All other columns will remain untouched. If None, all appropriate columns will be encoded. Defaults to None.
- **categories** (*list*) – A two dimensional list of categories, where *categories[i]* is a list of the categories for the column at index *i*. This can also be *None*, or *"auto"* if *top_n* is not None. Defaults to None.
- **drop** (*string, list*) – Method ("first" or "if_binary") to use to drop one category per feature. Can also be a list specifying which categories to drop for each feature. Defaults to 'if_binary'.
- **handle_unknown** (*string*) – Whether to ignore or error for unknown categories for a feature encountered during *fit* or *transform*. If either *top_n* or *categories* is used to limit the number of categories per column, this must be "ignore". Defaults to "ignore".
- **handle_missing** (*string*) – Options for how to handle missing (NaN) values encountered during *fit* or *transform*. If this is set to "as_category" and NaN values are within the *n* most frequent, "nan" values will be encoded as their own column. If this is set to "error", any missing values encountered will raise an error. Defaults to "error".
- **random_seed** (*int*) – Seed for the random number generator. Defaults to 0.

**Attributes**

| | |
|---|---|
| **hyper-parameter_ranges** | {} |
| **modifies_features** | True |
| **modifies_target** | False |
| **name** | One Hot Encoder |
| **training_only** | False |

**Methods**

| | |
|---|---|
| *categories* | Returns a list of the unique categories to be encoded for the particular feature, in order. |
| *clone* | Constructs a new component with the same parameters and random state. |
| *default_parameters* | Returns the default parameters for this component. |
| *describe* | Describe a component and its parameters. |
| *fit* | Fits the one-hot encoder component. |
| *fit_transform* | Fits on X and transforms X. |
| *get_feature_names* | Return feature names for the categorical features after fitting. |
| *load* | Loads component at file path. |
| *needs_fitting* | Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances. |
| *parameters* | Returns the parameters which were used to initialize the component. |
| *save* | Saves component at file path. |
| *transform* | One-hot encode the input data. |
| *update_parameters* | Updates the parameter dictionary of the component. |

**categories**(*self*, *feature_name*)

Returns a list of the unique categories to be encoded for the particular feature, in order.

**Parameters** **feature_name** (*str*) – The name of any feature provided to one-hot encoder during fit.

**Returns** The unique categories, in the same dtype as they were provided during fit.

**Return type** np.ndarray

**Raises** **ValueError** – If feature was not provided to one-hot encoder as a training feature.

**clone**(*self*)

Constructs a new component with the same parameters and random state.

**Returns** A new instance of this component with identical parameters and random state.

**default_parameters**(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

**Returns** Default parameters for this component.

**Return type** dict

**describe**(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

**Parameters**

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description as dictionary in the format {"name": name, "parameters": parameters}

**Returns** Returns dictionary if return_dict is True, else None.

**Return type** None or dict

`fit`(*self*, *X*, *y=None*)

Fits the one-hot encoder component.

**Parameters**

- **X** (*pd.DataFrame*) – The input training data of shape [n_samples, n_features].
- **y** (*pd.Series, optional*) – The target training data of length [n_samples].

**Returns** self

**Raises** **ValueError** – If encoding a column failed.

`fit_transform`(*self*, *X*, *y=None*)

Fits on X and transforms X.

**Parameters**

- **X** (*pd.DataFrame*) – Data to fit and transform.
- **y** (*pd.Series*) – Target data.

**Returns** Transformed X.

**Return type** pd.DataFrame

**Raises** **MethodPropertyNotFoundError** – If transformer does not have a transform method or a component_obj that implements transform.

`get_feature_names`(*self*)

Return feature names for the categorical features after fitting.

Feature names are formatted as {column name}_{category name}. In the event of a duplicate name, an integer will be added at the end of the feature name to distinguish it.

For example, consider a dataframe with a column called "A" and category "x_y" and another column called "A_x" with "y". In this example, the feature names would be "A_x_y" and "A_x_y_1".

**Returns** The feature names after encoding, provided in the same order as input_features.

**Return type** np.ndarray

`static load`(*file_path*)

Loads component at file path.

**Parameters** **file_path** (*str*) – Location to load file.

**Returns** ComponentBase object

`needs_fitting`(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need to be fit or whose fit methods do nothing.

**Returns** True.

`property parameters`(*self*)

Returns the parameters which were used to initialize the component.

**save**(*self*, *file_path*, *pickle_protocol=cloudpickle.DEFAULT_PROTOCOL*)

Saves component at file path.

**Parameters**

- **file_path** (*str*) – Location to save file.
- **pickle_protocol** (*int*) – The pickle data stream format.

**transform**(*self*, *X*, *y=None*)

One-hot encode the input data.

**Parameters**

- **X** (*pd.DataFrame*) – Features to one-hot encode.
- **y** (*pd.Series*) – Ignored.

**Returns** Transformed data, where each categorical feature has been encoded into numerical columns using one-hot encoding.

**Return type** pd.DataFrame

**update_parameters**(*self*, *update_dict*, *reset_fit=True*)

Updates the parameter dictionary of the component.

**Parameters**

- **update_dict** (*dict*) – A dict of parameters to update.
- **reset_fit** (*bool, optional*) – If True, will set *_is_fitted* to False.

**class** evalml.pipelines.components.transformers.encoders.**OrdinalEncoder**(*features_to_encode=None*, *categories=None*, *handle_unknown='error'*, *unknown_value=None*, *encoded_missing_value=None*, *random_seed=0*, ***kwargs*)

A transformer that encodes ordinal features as an array of ordinal integers representing the relative order of categories.

**Parameters**

- **features_to_encode** (*list[str]*) – List of columns to encode. All other columns will remain untouched. If None, all appropriate columns will be encoded. Defaults to None. The order of columns does not matter.
- **categories** (*dict[str, list[str]]*) – A dictionary mapping column names to their categories in the dataframes passed in at fit and transform. The order of categories specified for a column does not matter. Any category found in the data that is not present in categories will be handled as an unknown value. To not have unknown values raise an error, set handle_unknown to "use_encoded_value". Defaults to None.
- **handle_unknown** (*"error" or "use_encoded_value"*) – Whether to ignore or error for unknown categories for a feature encountered during *fit* or *transform*. When set to "error", an error will be raised when an unknown category is found. When set to "use_encoded_value", unknown categories will be encoded as the value given for the parameter unknown_value. Defaults to "error."

- **unknown_value** (*int or np.nan*) – The value to use for unknown categories seen during fit or transform. Required when the parameter handle_unknown is set to "use_encoded_value." The value has to be distinct from the values used to encode any of the categories in fit. Defaults to None.
- **encoded_missing_value** (*int or np.nan*) – The value to use for missing (null) values seen during fit or transform. Defaults to np.nan.
- **random_seed** (*int*) – Seed for the random number generator. Defaults to 0.

**Attributes**

| | |
|---|---|
| **hyper-parameter_ranges** | {} |
| **modifies_features** | True |
| **modifies_target** | False |
| **name** | Ordinal Encoder |
| **training_only** | False |

**Methods**

| | |
|---|---|
| *categories* | Returns a list of the unique categories to be encoded for the particular feature, in order. |
| *clone* | Constructs a new component with the same parameters and random state. |
| *default_parameters* | Returns the default parameters for this component. |
| *describe* | Describe a component and its parameters. |
| *fit* | Fits the ordinal encoder component. |
| *fit_transform* | Fits on X and transforms X. |
| *get_feature_names* | Return feature names for the ordinal features after fitting. |
| *load* | Loads component at file path. |
| *needs_fitting* | Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances. |
| *parameters* | Returns the parameters which were used to initialize the component. |
| *save* | Saves component at file path. |
| *transform* | Ordinally encode the input data. |
| *update_parameters* | Updates the parameter dictionary of the component. |

**categories**(*self*, *feature_name*)

Returns a list of the unique categories to be encoded for the particular feature, in order.

**Parameters** **feature_name** (*str*) – The name of any feature provided to ordinal encoder during fit.

**Returns** The unique categories, in the same dtype as they were provided during fit.

**Return type** np.ndarray

**Raises** **ValueError** – If feature was not provided to ordinal encoder as a training feature.

**clone**(*self*)

Constructs a new component with the same parameters and random state.

**Returns** A new instance of this component with identical parameters and random state.

**default_parameters**(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

**Returns** Default parameters for this component.

**Return type** dict

**describe**(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

**Parameters**

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description as dictionary in the format {"name": name, "parameters": parameters}

**Returns** Returns dictionary if return_dict is True, else None.

**Return type** None or dict

**fit**(*self*, *X*, *y=None*)

Fits the ordinal encoder component.

**Parameters**

- **X** (*pd.DataFrame*) – The input training data of shape [n_samples, n_features].
- **y** (*pd.Series, optional*) – The target training data of length [n_samples].

**Returns** self

**Raises**

- **ValueError** – If encoding a column failed.
- **TypeError** – If non-Ordinal columns are specified in features_to_encode.

**fit_transform**(*self*, *X*, *y=None*)

Fits on X and transforms X.

**Parameters**

- **X** (*pd.DataFrame*) – Data to fit and transform.
- **y** (*pd.Series*) – Target data.

**Returns** Transformed X.

**Return type** pd.DataFrame

**Raises** **MethodPropertyNotFoundError** – If transformer does not have a transform method or a component_obj that implements transform.

**get_feature_names**(*self*)

Return feature names for the ordinal features after fitting.

Feature names are formatted as {column name}_ordinal_encoding.

**Returns** The feature names after encoding, provided in the same order as input_features.

**Return type** np.ndarray

**static load**(*file_path*)

Loads component at file path.

**Parameters** **file_path** (*str*) – Location to load file.

**Returns** ComponentBase object

**needs_fitting**(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need to be fit or whose fit methods do nothing.

**Returns** True.

**property parameters**(*self*)

Returns the parameters which were used to initialize the component.

**save**(*self*, *file_path*, *pickle_protocol=cloudpickle.DEFAULT_PROTOCOL*)

Saves component at file path.

**Parameters**

- **file_path** (*str*) – Location to save file.
- **pickle_protocol** (*int*) – The pickle data stream format.

**transform**(*self*, *X*, *y=None*)

Ordinally encode the input data.

**Parameters**

- **X** (*pd.DataFrame*) – Features to encode.
- **y** (*pd.Series*) – Ignored.

**Returns** Transformed data, where each ordinal feature has been encoded into a numerical column where ordinal integers represent the relative order of categories.

**Return type** pd.DataFrame

**update_parameters**(*self*, *update_dict*, *reset_fit=True*)

Updates the parameter dictionary of the component.

**Parameters**

- **update_dict** (*dict*) – A dict of parameters to update.
- **reset_fit** (*bool, optional*) – If True, will set *_is_fitted* to False.

**class** evalml.pipelines.components.transformers.encoders.**TargetEncoder**(*cols=None*, *smoothing=1*, *handle_unknown='value'*, *handle_missing='value'*, *random_seed=0*, ***kwargs*)

A transformer that encodes categorical features into target encodings.

**Parameters**

- **cols** (*list*) – Columns to encode. If None, all string columns will be encoded, otherwise only the columns provided will be encoded. Defaults to None

- **smoothing** (*float*) – The smoothing factor to apply. The larger this value is, the more influence the expected target value has on the resulting target encodings. Must be strictly larger than 0. Defaults to 1.0
- **handle_unknown** (*string*) – Determines how to handle unknown categories for a feature encountered. Options are 'value', 'error', nd 'return_nan'. Defaults to 'value', which replaces with the target mean
- **handle_missing** (*string*) – Determines how to handle missing values encountered during *fit* or *transform*. Options are 'value', 'error', and 'return_nan'. Defaults to 'value', which replaces with the target mean
- **random_seed** (*int*) – Seed for the random number generator. Defaults to 0.

**Attributes**

| | |
|---|---|
| **hyper-parameter_ranges** | {} |
| **modifies_features** | True |
| **modifies_target** | False |
| **name** | Target Encoder |
| **training_only** | False |

**Methods**

| | |
|---|---|
| *clone* | Constructs a new component with the same parameters and random state. |
| *default_parameters* | Returns the default parameters for this component. |
| *describe* | Describe a component and its parameters. |
| *fit* | Fits the target encoder. |
| *fit_transform* | Fit and transform data using the target encoder. |
| *get_feature_names* | Return feature names for the input features after fitting. |
| *load* | Loads component at file path. |
| *needs_fitting* | Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances. |
| *parameters* | Returns the parameters which were used to initialize the component. |
| *save* | Saves component at file path. |
| *transform* | Transform data using the fitted target encoder. |
| *update_parameters* | Updates the parameter dictionary of the component. |

**clone**(*self*)

Constructs a new component with the same parameters and random state.

**Returns** A new instance of this component with identical parameters and random state.

**default_parameters**(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

**Returns** Default parameters for this component.

**Return type** dict

`describe`(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

**Parameters**

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description as dictionary in the format {"name": name, "parameters": parameters}

**Returns** Returns dictionary if return_dict is True, else None.

**Return type** None or dict

`fit`(*self*, *X*, *y*)

Fits the target encoder.

**Parameters**

- **X** (*pd.DataFrame*) – The input training data of shape [n_samples, n_features].
- **y** (*pd.Series, optional*) – The target training data of length [n_samples].

**Returns** self

`fit_transform`(*self*, *X*, *y*)

Fit and transform data using the target encoder.

**Parameters**

- **X** (*pd.DataFrame*) – The input training data of shape [n_samples, n_features].
- **y** (*pd.Series, optional*) – The target training data of length [n_samples].

**Returns** Transformed data.

**Return type** pd.DataFrame

`get_feature_names`(*self*)

Return feature names for the input features after fitting.

**Returns** The feature names after encoding.

**Return type** np.array

`static load`(*file_path*)

Loads component at file path.

**Parameters** **file_path** (*str*) – Location to load file.

**Returns** ComponentBase object

`needs_fitting`(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need to be fit or whose fit methods do nothing.

**Returns** True.

`property parameters`(*self*)

Returns the parameters which were used to initialize the component.

**save**(*self*, *file_path*, *pickle_protocol=cloudpickle.DEFAULT_PROTOCOL*)

Saves component at file path.

**Parameters**

- **file_path** (*str*) – Location to save file.
- **pickle_protocol** (*int*) – The pickle data stream format.

**transform**(*self*, *X*, *y=None*)

Transform data using the fitted target encoder.

**Parameters**

- **X** (*pd.DataFrame*) – The input training data of shape [n_samples, n_features].
- **y** (*pd.Series, optional*) – The target training data of length [n_samples].

**Returns** Transformed data.

**Return type** pd.DataFrame

**update_parameters**(*self*, *update_dict*, *reset_fit=True*)

Updates the parameter dictionary of the component.

**Parameters**

- **update_dict** (*dict*) – A dict of parameters to update.
- **reset_fit** (*bool, optional*) – If True, will set *_is_fitted* to False.

## feature_selection

Components that select features.

### Submodules

### feature_selector

Component that selects top features based on importance weights.

### Module Contents

### Classes Summary

| | |
|---|---|
| *FeatureSelector* | Selects top features based on importance weights. |

## Contents

**class** evalml.pipelines.components.transformers.feature_selection.feature_selector.**FeatureSelector**(*parame com- po- nent_o ran- dom_se **kwar*

Selects top features based on importance weights.

**Parameters**

- **parameters** (*dict*) – Dictionary of parameters for the component. Defaults to None.
- **component_obj** (*obj*) – Third-party objects useful in component implementation. Defaults to None.
- **random_seed** (*int*) – Seed for the random number generator. Defaults to 0.

**Attributes**

| | |
|---|---|
| **modifies_features** | True |
| **modifies_target** | False |
| **training_only** | False |

**Methods**

| | |
|---|---|
| *clone* | Constructs a new component with the same parameters and random state. |
| *default_parameters* | Returns the default parameters for this component. |
| *describe* | Describe a component and its parameters. |
| *fit* | Fits component to data. |
| *fit_transform* | Fit and transform data using the feature selector. |
| *get_names* | Get names of selected features. |
| *load* | Loads component at file path. |
| *name* | Returns string name of this component. |
| *needs_fitting* | Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances. |
| *parameters* | Returns the parameters which were used to initialize the component. |
| *save* | Saves component at file path. |
| *transform* | Transforms input data by selecting features. If the component_obj does not have a transform method, will raise an MethodPropertyNotFoundError exception. |
| *update_parameters* | Updates the parameter dictionary of the component. |

**clone**(*self*)

Constructs a new component with the same parameters and random state.

**Returns** A new instance of this component with identical parameters and random state.

`default_parameters`(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

**Returns** Default parameters for this component.

**Return type** dict

`describe`(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

**Parameters**

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description as dictionary in the format {"name": name, "parameters": parameters}

**Returns** Returns dictionary if return_dict is True, else None.

**Return type** None or dict

`fit`(*self*, *X*, *y=None*)

Fits component to data.

**Parameters**

- **X** (*pd.DataFrame*) – The input training data of shape [n_samples, n_features]
- **y** (*pd.Series, optional*) – The target training data of length [n_samples]

**Returns** self

**Raises** **MethodPropertyNotFoundError** – If component does not have a fit method or a component_obj that implements fit.

`fit_transform`(*self*, *X*, *y=None*)

Fit and transform data using the feature selector.

**Parameters**

- **X** (*pd.DataFrame*) – The input training data of shape [n_samples, n_features].
- **y** (*pd.Series, optional*) – The target training data of length [n_samples].

**Returns** Transformed data.

**Return type** pd.DataFrame

`get_names`(*self*)

Get names of selected features.

**Returns** List of the names of features selected.

**Return type** list[str]

`static load`(*file_path*)

Loads component at file path.

**Parameters** **file_path** (*str*) – Location to load file.

**Returns** ComponentBase object

**property name**(*cls*)

Returns string name of this component.

**needs_fitting**(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need to be fit or whose fit methods do nothing.

**Returns** True.

**property parameters**(*self*)

Returns the parameters which were used to initialize the component.

**save**(*self*, *file_path*, *pickle_protocol=cloudpickle.DEFAULT_PROTOCOL*)

Saves component at file path.

**Parameters**

- **file_path** (*str*) – Location to save file.
- **pickle_protocol** (*int*) – The pickle data stream format.

**transform**(*self*, *X*, *y=None*)

Transforms input data by selecting features. If the component_obj does not have a transform method, will raise an MethodPropertyNotFoundError exception.

**Parameters**

- **X** (*pd.DataFrame*) – Data to transform.
- **y** (*pd.Series, optional*) – Target data. Ignored.

**Returns** Transformed X

**Return type** pd.DataFrame

**Raises** **MethodPropertyNotFoundError** – If feature selector does not have a transform method or a component_obj that implements transform

**update_parameters**(*self*, *update_dict*, *reset_fit=True*)

Updates the parameter dictionary of the component.

**Parameters**

- **update_dict** (*dict*) – A dict of parameters to update.
- **reset_fit** (*bool, optional*) – If True, will set *_is_fitted* to False.

### recursive_feature_elimination_selector

Components that select top features based on recursive feature elimination with a Random Forest model.

### Module Contents

### Classes Summary

| | |
|---|---|
| *RecursiveFeatureEliminationSelector* | Selects relevant features using recursive feature elimination. |
| *RFClassifierRFESelector* | Selects relevant features using recursive feature elimination with a Random Forest Classifier. |
| *RFRegressorRFESelector* | Selects relevant features using recursive feature elimination with a Random Forest Regressor. |

### Contents

**class** `evalml.pipelines.components.transformers.feature_selection.recursive_feature_elimination_selector`

Selects relevant features using recursive feature elimination.

**Attributes**

| | |
|---|---|
| **hyper-parameter_ranges** | { "step": Real(0.05, 0.25)} |
| **modifies_features** | True |
| **modifies_target** | False |
| **training_only** | False |

**Methods**

| | |
|---|---|
| *clone* | Constructs a new component with the same parameters and random state. |
| *default_parameters* | Returns the default parameters for this component. |
| *describe* | Describe a component and its parameters. |
| *fit* | Fits component to data. |
| *fit_transform* | Fit and transform data using the feature selector. |
| *get_names* | Get names of selected features. |
| *load* | Loads component at file path. |
| *name* | Returns string name of this component. |
| *needs_fitting* | Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances. |
| *parameters* | Returns the parameters which were used to initialize the component. |
| *save* | Saves component at file path. |
| *transform* | Transforms input data by selecting features. If the component_obj does not have a transform method, will raise an MethodPropertyNotFoundError exception. |
| *update_parameters* | Updates the parameter dictionary of the component. |

**clone**(*self*)

Constructs a new component with the same parameters and random state.

**Returns** A new instance of this component with identical parameters and random state.

**default_parameters**(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

**Returns** Default parameters for this component.

**Return type** dict

**describe**(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

**Parameters**

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description as dictionary in the format {"name": name, "parameters": parameters}

**Returns** Returns dictionary if return_dict is True, else None.

**Return type** None or dict

**fit**(*self*, *X*, *y=None*)

Fits component to data.

**Parameters**

- **X** (*pd.DataFrame*) – The input training data of shape [n_samples, n_features]
- **y** (*pd.Series, optional*) – The target training data of length [n_samples]

**Returns** self

**Raises** `MethodPropertyNotFoundError` – If component does not have a fit method or a component_obj that implements fit.

`fit_transform`(*self*, *X*, *y=None*)

Fit and transform data using the feature selector.

**Parameters**

- `X` (*pd.DataFrame*) – The input training data of shape [n_samples, n_features].
- `y` (*pd.Series, optional*) – The target training data of length [n_samples].

**Returns** Transformed data.

**Return type** pd.DataFrame

`get_names`(*self*)

Get names of selected features.

**Returns** List of the names of features selected.

**Return type** list[str]

`static load`(*file_path*)

Loads component at file path.

**Parameters** `file_path` (*str*) – Location to load file.

**Returns** ComponentBase object

`property name`(*cls*)

Returns string name of this component.

`needs_fitting`(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need to be fit or whose fit methods do nothing.

**Returns** True.

`property parameters`(*self*)

Returns the parameters which were used to initialize the component.

`save`(*self*, *file_path*, *pickle_protocol=cloudpickle.DEFAULT_PROTOCOL*)

Saves component at file path.

**Parameters**

- `file_path` (*str*) – Location to save file.
- `pickle_protocol` (*int*) – The pickle data stream format.

`transform`(*self*, *X*, *y=None*)

Transforms input data by selecting features. If the component_obj does not have a transform method, will raise an MethodPropertyNotFoundError exception.

**Parameters**

- `X` (*pd.DataFrame*) – Data to transform.
- `y` (*pd.Series, optional*) – Target data. Ignored.

**Returns** Transformed X

**Return type** pd.DataFrame

**Raises** **MethodPropertyNotFoundError** – If feature selector does not have a transform method or a component_obj that implements transform

**update_parameters**(*self*, *update_dict*, *reset_fit=True*)

Updates the parameter dictionary of the component.

**Parameters**

- **update_dict** (*dict*) – A dict of parameters to update.
- **reset_fit** (*bool, optional*) – If True, will set *_is_fitted* to False.

**class** evalml.pipelines.components.transformers.feature_selection.recursive_feature_elimination_selector

Selects relevant features using recursive feature elimination with a Random Forest Classifier.

**Parameters**

- **step** (*int, float*) – The number of features to eliminate in each iteration. If an integer is specified this will represent the number of features to eliminate. If a float is specified this represents the percentage of features to eliminate each iteration. The last iteration may drop fewer than this number of features in order to satisfy the min_features_to_select constraint. Defaults to 0.2.
- **min_features_to_select** (*int*) – The minimum number of features to return. Defaults to 1.
- **cv** (*int or None*) – Number of folds to use for the cross-validation splitting strategy. Defaults to None which will use 5 folds.
- **scoring** (*str, callable or None*) – A string or scorer callable object to specify the scoring method.
- **n_jobs** (*int or None*) – Number of jobs to run in parallel. -1 uses all processes. Defaults to -1.
- **n_estimators** (*int*) – The number of trees in the forest. Defaults to 10.
- **max_depth** (*int*) – Maximum tree depth for base learners. Defaults to None.
- **random_seed** (*int*) – Seed for the random number generator. Defaults to 0.

**Attributes**

| | |
|---|---|
| **hyper-parameter_ranges** | { "step": Real(0.05, 0.25)} |
| **modifies_features** | True |
| **modifies_target** | False |
| **name** | RFE Selector with RF Classifier |
| **training_only** | False |

**Methods**

| | |
|---|---|
| *clone* | Constructs a new component with the same parameters and random state. |
| *default_parameters* | Returns the default parameters for this component. |
| *describe* | Describe a component and its parameters. |
| *fit* | Fits component to data. |
| *fit_transform* | Fit and transform data using the feature selector. |
| *get_names* | Get names of selected features. |
| *load* | Loads component at file path. |
| *needs_fitting* | Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances. |
| *parameters* | Returns the parameters which were used to initialize the component. |
| *save* | Saves component at file path. |
| *transform* | Transforms input data by selecting features. If the component_obj does not have a transform method, will raise an MethodPropertyNotFoundError exception. |
| *update_parameters* | Updates the parameter dictionary of the component. |

**clone**(*self*)

Constructs a new component with the same parameters and random state.

**Returns** A new instance of this component with identical parameters and random state.

**default_parameters**(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

**Returns** Default parameters for this component.

**Return type** dict

**describe**(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

**Parameters**

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description as dictionary in the format {"name": name, "parameters": parameters}

**Returns** Returns dictionary if return_dict is True, else None.

**Return type** None or dict

`fit`(*self*, *X*, *y=None*)

Fits component to data.

**Parameters**

- **X** (*pd.DataFrame*) – The input training data of shape [n_samples, n_features]
- **y** (*pd.Series, optional*) – The target training data of length [n_samples]

**Returns** self

**Raises** **MethodPropertyNotFoundError** – If component does not have a fit method or a component_obj that implements fit.

`fit_transform`(*self*, *X*, *y=None*)

Fit and transform data using the feature selector.

**Parameters**

- **X** (*pd.DataFrame*) – The input training data of shape [n_samples, n_features].
- **y** (*pd.Series, optional*) – The target training data of length [n_samples].

**Returns** Transformed data.

**Return type** pd.DataFrame

`get_names`(*self*)

Get names of selected features.

**Returns** List of the names of features selected.

**Return type** list[str]

`static load`(*file_path*)

Loads component at file path.

**Parameters** **file_path** (*str*) – Location to load file.

**Returns** ComponentBase object

`needs_fitting`(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need to be fit or whose fit methods do nothing.

**Returns** True.

`property parameters`(*self*)

Returns the parameters which were used to initialize the component.

`save`(*self*, *file_path*, *pickle_protocol=cloudpickle.DEFAULT_PROTOCOL*)

Saves component at file path.

**Parameters**

- **file_path** (*str*) – Location to save file.
- **pickle_protocol** (*int*) – The pickle data stream format.

**transform**(*self*, *X*, *y=None*)

Transforms input data by selecting features. If the component_obj does not have a transform method, will raise an MethodPropertyNotFoundError exception.

**Parameters**

- **X** (*pd.DataFrame*) – Data to transform.
- **y** (*pd.Series, optional*) – Target data. Ignored.

**Returns** Transformed X

**Return type** pd.DataFrame

**Raises** **MethodPropertyNotFoundError** – If feature selector does not have a transform method or a component_obj that implements transform

**update_parameters**(*self*, *update_dict*, *reset_fit=True*)

Updates the parameter dictionary of the component.

**Parameters**

- **update_dict** (*dict*) – A dict of parameters to update.
- **reset_fit** (*bool, optional*) – If True, will set *_is_fitted* to False.

**class** evalml.pipelines.components.transformers.feature_selection.recursive_feature_elimination_selector

Selects relevant features using recursive feature elimination with a Random Forest Regressor.

**Parameters**

- **step** (*int, float*) – The number of features to eliminate in each iteration. If an integer is specified this will represent the number of features to eliminate. If a float is specified this represents the percentage of features to eliminate each iteration. The last iteration may drop fewer than this number of features in order to satisfy the min_features_to_select constraint. Defaults to 0.2.
- **min_features_to_select** (*int*) – The minimum number of features to return. Defaults to 1.
- **cv** (*int or None*) – Number of folds to use for the cross-validation splitting strategy. Defaults to None which will use 5 folds.
- **scoring** (*str, callable or None*) – A string or scorer callable object to specify the scoring method.
- **n_jobs** (*int or None*) – Number of jobs to run in parallel. -1 uses all processes. Defaults to -1.
- **n_estimators** (*int*) – The number of trees in the forest. Defaults to 10.

- **max_depth** (*int*) – Maximum tree depth for base learners. Defaults to None.
- **random_seed** (*int*) – Seed for the random number generator. Defaults to 0.

**Attributes**

| | |
|---|---|
| **hyper-parameter_ranges** | { "step": Real(0.05, 0.25)} |
| **modifies_features** | True |
| **modifies_target** | False |
| **name** | RFE Selector with RF Regressor |
| **training_only** | False |

**Methods**

| | |
|---|---|
| *clone* | Constructs a new component with the same parameters and random state. |
| *default_parameters* | Returns the default parameters for this component. |
| *describe* | Describe a component and its parameters. |
| *fit* | Fits component to data. |
| *fit_transform* | Fit and transform data using the feature selector. |
| *get_names* | Get names of selected features. |
| *load* | Loads component at file path. |
| *needs_fitting* | Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances. |
| *parameters* | Returns the parameters which were used to initialize the component. |
| *save* | Saves component at file path. |
| *transform* | Transforms input data by selecting features. If the component_obj does not have a transform method, will raise an MethodPropertyNotFoundError exception. |
| *update_parameters* | Updates the parameter dictionary of the component. |

**clone**(*self*)

Constructs a new component with the same parameters and random state.

**Returns** A new instance of this component with identical parameters and random state.

**default_parameters**(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

**Returns** Default parameters for this component.

**Return type** dict

**describe**(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

**Parameters**

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description as dictionary in the format {"name": name, "parameters": parameters}

**Returns** Returns dictionary if return_dict is True, else None.

**Return type** None or dict

**fit**(*self*, *X*, *y=None*)

Fits component to data.

**Parameters**

- **X** (*pd.DataFrame*) – The input training data of shape [n_samples, n_features]
- **y** (*pd.Series, optional*) – The target training data of length [n_samples]

**Returns** self

**Raises** **MethodPropertyNotFoundError** – If component does not have a fit method or a component_obj that implements fit.

**fit_transform**(*self*, *X*, *y=None*)

Fit and transform data using the feature selector.

**Parameters**

- **X** (*pd.DataFrame*) – The input training data of shape [n_samples, n_features].
- **y** (*pd.Series, optional*) – The target training data of length [n_samples].

**Returns** Transformed data.

**Return type** pd.DataFrame

**get_names**(*self*)

Get names of selected features.

**Returns** List of the names of features selected.

**Return type** list[str]

**static load**(*file_path*)

Loads component at file path.

**Parameters** **file_path** (*str*) – Location to load file.

**Returns** ComponentBase object

**needs_fitting**(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need to be fit or whose fit methods do nothing.

**Returns** True.

**property parameters**(*self*)

Returns the parameters which were used to initialize the component.

**save**(*self*, *file_path*, *pickle_protocol=cloudpickle.DEFAULT_PROTOCOL*)

Saves component at file path.

**Parameters**

- **file_path** (*str*) – Location to save file.
- **pickle_protocol** (*int*) – The pickle data stream format.

**transform**(*self*, *X*, *y=None*)

Transforms input data by selecting features. If the component_obj does not have a transform method, will raise an MethodPropertyNotFoundError exception.

**Parameters**

- **X** (*pd.DataFrame*) – Data to transform.
- **y** (*pd.Series, optional*) – Target data. Ignored.

**Returns** Transformed X

**Return type** pd.DataFrame

**Raises** **MethodPropertyNotFoundError** – If feature selector does not have a transform method or a component_obj that implements transform

**update_parameters**(*self*, *update_dict*, *reset_fit=True*)

Updates the parameter dictionary of the component.

**Parameters**

- **update_dict** (*dict*) – A dict of parameters to update.
- **reset_fit** (*bool, optional*) – If True, will set *_is_fitted* to False.

### rf_classifier_feature_selector

Component that selects top features based on importance weights using a Random Forest classifier.

### Module Contents

### Classes Summary

| | |
|---|---|
| *RFClassifierSelectFromModel* | Selects top features based on importance weights using a Random Forest classifier. |

### Contents

**class** evalml.pipelines.components.transformers.feature_selection.rf_classifier_feature_selector.**RFClassi**

Selects top features based on importance weights using a Random Forest classifier.

**Parameters**

- **number_features** (*int*) – The maximum number of features to select. If both percent_features and number_features are specified, take the greater number of features. Defaults to None.
- **n_estimators** (*int*) – The number of trees in the forest. Defaults to 10.
- **max_depth** (*int*) – Maximum tree depth for base learners. Defaults to None.
- **percent_features** (*float*) – Percentage of features to use. If both percent_features and number_features are specified, take the greater number of features. Defaults to 0.5.
- **threshold** (*string or float*) – The threshold value to use for feature selection. Features whose importance is greater or equal are kept while the others are discarded. If "median", then the threshold value is the median of the feature importances. A scaling factor (e.g., "1.25*mean") may also be used. Defaults to median.
- **n_jobs** (*int or None*) – Number of jobs to run in parallel. -1 uses all processes. Defaults to -1.
- **random_seed** (*int*) – Seed for the random number generator. Defaults to 0.

**Attributes**

| | |
|---|---|
| **hyperparameter_ranges** | { "percent_features": Real(0.01, 1), "threshold": ["mean", "median"],} |
| **modifies_features** | True |
| **modifies_target** | False |
| **name** | RF Classifier Select From Model |
| **training_only** | False |

**Methods**

| | |
|---|---|
| *clone* | Constructs a new component with the same parameters and random state. |
| *default_parameters* | Returns the default parameters for this component. |
| *describe* | Describe a component and its parameters. |
| *fit* | Fits component to data. |
| *fit_transform* | Fit and transform data using the feature selector. |
| *get_names* | Get names of selected features. |
| *load* | Loads component at file path. |
| *needs_fitting* | Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances. |
| *parameters* | Returns the parameters which were used to initialize the component. |
| *save* | Saves component at file path. |
| *transform* | Transforms input data by selecting features. If the component_obj does not have a transform method, will raise an MethodPropertyNotFoundError exception. |
| *update_parameters* | Updates the parameter dictionary of the component. |

**clone**(*self*)

Constructs a new component with the same parameters and random state.

**Returns** A new instance of this component with identical parameters and random state.

**default_parameters**(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

**Returns** Default parameters for this component.

**Return type** dict

**describe**(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

**Parameters**

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description as dictionary in the format {"name": name, "parameters": parameters}

**Returns** Returns dictionary if return_dict is True, else None.

**Return type** None or dict

**fit**(*self*, *X*, *y=None*)

Fits component to data.

**Parameters**

- **X** (*pd.DataFrame*) – The input training data of shape [n_samples, n_features]
- **y** (*pd.Series, optional*) – The target training data of length [n_samples]

**Returns** self

**Raises** `MethodPropertyNotFoundError` – If component does not have a fit method or a component_obj that implements fit.

`fit_transform`(*self*, *X*, *y=None*)

Fit and transform data using the feature selector.

**Parameters**

- `X` (*pd.DataFrame*) – The input training data of shape [n_samples, n_features].
- `y` (*pd.Series, optional*) – The target training data of length [n_samples].

**Returns** Transformed data.

**Return type** pd.DataFrame

`get_names`(*self*)

Get names of selected features.

**Returns** List of the names of features selected.

**Return type** list[str]

`static load`(*file_path*)

Loads component at file path.

**Parameters** `file_path` (*str*) – Location to load file.

**Returns** ComponentBase object

`needs_fitting`(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need to be fit or whose fit methods do nothing.

**Returns** True.

`property parameters`(*self*)

Returns the parameters which were used to initialize the component.

`save`(*self*, *file_path*, *pickle_protocol=cloudpickle.DEFAULT_PROTOCOL*)

Saves component at file path.

**Parameters**

- `file_path` (*str*) – Location to save file.
- `pickle_protocol` (*int*) – The pickle data stream format.

`transform`(*self*, *X*, *y=None*)

Transforms input data by selecting features. If the component_obj does not have a transform method, will raise an MethodPropertyNotFoundError exception.

**Parameters**

- `X` (*pd.DataFrame*) – Data to transform.
- `y` (*pd.Series, optional*) – Target data. Ignored.

**Returns** Transformed X

**Return type** pd.DataFrame

**Raises** `MethodPropertyNotFoundError` – If feature selector does not have a transform method or a component_obj that implements transform

**update_parameters**(*self*, *update_dict*, *reset_fit=True*)

Updates the parameter dictionary of the component.

**Parameters**

- **update_dict** (*dict*) – A dict of parameters to update.
- **reset_fit** (*bool, optional*) – If True, will set *_is_fitted* to False.

### rf_regressor_feature_selector

Component that selects top features based on importance weights using a Random Forest regresor.

### Module Contents

### Classes Summary

| | |
|---|---|
| *RFRegressorSelectFromModel* | Selects top features based on importance weights using a Random Forest regressor. |

### Contents

**class** evalml.pipelines.components.transformers.feature_selection.rf_regressor_feature_selector.**RFRegress**

Selects top features based on importance weights using a Random Forest regressor.

**Parameters**

- **number_features** (*int*) – The maximum number of features to select. If both percent_features and number_features are specified, take the greater number of features. Defaults to 0.5.
- **n_estimators** (*int*) – The number of trees in the forest. Defaults to 10.
- **max_depth** (*int*) – Maximum tree depth for base learners. Defaults to None.
- **percent_features** (*float*) – Percentage of features to use. If both percent_features and number_features are specified, take the greater number of features. Defaults to 0.5.
- **threshold** (*string or float*) – The threshold value to use for feature selection. Features whose importance is greater or equal are kept while the others are discarded. If "median", then the threshold value is the median of the feature importances. A scaling factor (e.g., "1.25*mean") may also be used. Defaults to median.

- **n_jobs** (*int or None*) – Number of jobs to run in parallel. -1 uses all processes. Defaults to -1.
- **random_seed** (*int*) – Seed for the random number generator. Defaults to 0.

**Attributes**

| | |
|---|---|
| **hyper-parameter_ranges** | { "percent_features": Real(0.01, 1), "threshold": ["mean", "median"],} |
| **modifies_features** | True |
| **modifies_target** | False |
| **name** | RF Regressor Select From Model |
| **training_only** | False |

**Methods**

| | |
|---|---|
| *clone* | Constructs a new component with the same parameters and random state. |
| *default_parameters* | Returns the default parameters for this component. |
| *describe* | Describe a component and its parameters. |
| *fit* | Fits component to data. |
| *fit_transform* | Fit and transform data using the feature selector. |
| *get_names* | Get names of selected features. |
| *load* | Loads component at file path. |
| *needs_fitting* | Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances. |
| *parameters* | Returns the parameters which were used to initialize the component. |
| *save* | Saves component at file path. |
| *transform* | Transforms input data by selecting features. If the component_obj does not have a transform method, will raise an MethodPropertyNotFoundError exception. |
| *update_parameters* | Updates the parameter dictionary of the component. |

**clone**(*self*)

Constructs a new component with the same parameters and random state.

**Returns** A new instance of this component with identical parameters and random state.

**default_parameters**(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

**Returns** Default parameters for this component.

**Return type** dict

**describe**(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

**Parameters**

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description as dictionary in the format {"name": name, "parameters": parameters}

**Returns** Returns dictionary if return_dict is True, else None.

**Return type** None or dict

**fit**(*self*, *X*, *y=None*)

Fits component to data.

**Parameters**

- **X** (*pd.DataFrame*) – The input training data of shape [n_samples, n_features]
- **y** (*pd.Series, optional*) – The target training data of length [n_samples]

**Returns** self

**Raises** **MethodPropertyNotFoundError** – If component does not have a fit method or a component_obj that implements fit.

**fit_transform**(*self*, *X*, *y=None*)

Fit and transform data using the feature selector.

**Parameters**

- **X** (*pd.DataFrame*) – The input training data of shape [n_samples, n_features].
- **y** (*pd.Series, optional*) – The target training data of length [n_samples].

**Returns** Transformed data.

**Return type** pd.DataFrame

**get_names**(*self*)

Get names of selected features.

**Returns** List of the names of features selected.

**Return type** list[str]

**static load**(*file_path*)

Loads component at file path.

**Parameters** **file_path** (*str*) – Location to load file.

**Returns** ComponentBase object

**needs_fitting**(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need to be fit or whose fit methods do nothing.

**Returns** True.

**property parameters**(*self*)

Returns the parameters which were used to initialize the component.

**save**(*self*, *file_path*, *pickle_protocol=cloudpickle.DEFAULT_PROTOCOL*)

Saves component at file path.

**Parameters**

- **file_path** (*str*) – Location to save file.
- **pickle_protocol** (*int*) – The pickle data stream format.

**transform**(*self*, *X*, *y=None*)

Transforms input data by selecting features. If the component_obj does not have a transform method, will raise an MethodPropertyNotFoundError exception.

**Parameters**

- **X** (*pd.DataFrame*) – Data to transform.
- **y** (*pd.Series, optional*) – Target data. Ignored.

**Returns** Transformed X

**Return type** pd.DataFrame

**Raises** **MethodPropertyNotFoundError** – If feature selector does not have a transform method or a component_obj that implements transform

**update_parameters**(*self*, *update_dict*, *reset_fit=True*)

Updates the parameter dictionary of the component.

**Parameters**

- **update_dict** (*dict*) – A dict of parameters to update.
- **reset_fit** (*bool, optional*) – If True, will set *_is_fitted* to False.

## Package Contents

## Classes Summary

| | |
|---|---|
| *FeatureSelector* | Selects top features based on importance weights. |
| *RFClassifierRFESelector* | Selects relevant features using recursive feature elimination with a Random Forest Classifier. |
| *RFClassifierSelectFromModel* | Selects top features based on importance weights using a Random Forest classifier. |
| *RFRegressorRFESelector* | Selects relevant features using recursive feature elimination with a Random Forest Regressor. |
| *RFRegressorSelectFromModel* | Selects top features based on importance weights using a Random Forest regressor. |

## Contents

**class** evalml.pipelines.components.transformers.feature_selection.**FeatureSelector**(*parameters=None, component_obj=None, random_seed=0, **kwargs*)

Selects top features based on importance weights.

**Parameters**

- **parameters** (*dict*) – Dictionary of parameters for the component. Defaults to None.
- **component_obj** (*obj*) – Third-party objects useful in component implementation. Defaults to None.
- **random_seed** (*int*) – Seed for the random number generator. Defaults to 0.

**Attributes**

| | |
|---|---|
| **modifies_features** | True |
| **modifies_target** | False |
| **training_only** | False |

**Methods**

| | |
|---|---|
| *clone* | Constructs a new component with the same parameters and random state. |
| *default_parameters* | Returns the default parameters for this component. |
| *describe* | Describe a component and its parameters. |
| *fit* | Fits component to data. |
| *fit_transform* | Fit and transform data using the feature selector. |
| *get_names* | Get names of selected features. |
| *load* | Loads component at file path. |
| *name* | Returns string name of this component. |
| *needs_fitting* | Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances. |
| *parameters* | Returns the parameters which were used to initialize the component. |
| *save* | Saves component at file path. |
| *transform* | Transforms input data by selecting features. If the component_obj does not have a transform method, will raise an MethodPropertyNotFoundError exception. |
| *update_parameters* | Updates the parameter dictionary of the component. |

**clone**(*self*)

Constructs a new component with the same parameters and random state.

**Returns** A new instance of this component with identical parameters and random state.

**default_parameters**(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

**Returns** Default parameters for this component.

**Return type** dict

**describe**(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

**Parameters**

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description as dictionary in the format {"name": name, "parameters": parameters}

**Returns** Returns dictionary if return_dict is True, else None.

**Return type** None or dict

**fit**(*self*, *X*, *y=None*)

Fits component to data.

**Parameters**

- **X** (*pd.DataFrame*) – The input training data of shape [n_samples, n_features]
- **y** (*pd.Series, optional*) – The target training data of length [n_samples]

**Returns** self

**Raises** **MethodPropertyNotFoundError** – If component does not have a fit method or a component_obj that implements fit.

**fit_transform**(*self*, *X*, *y=None*)

Fit and transform data using the feature selector.

**Parameters**

- **X** (*pd.DataFrame*) – The input training data of shape [n_samples, n_features].
- **y** (*pd.Series, optional*) – The target training data of length [n_samples].

**Returns** Transformed data.

**Return type** pd.DataFrame

**get_names**(*self*)

Get names of selected features.

**Returns** List of the names of features selected.

**Return type** list[str]

**static load**(*file_path*)

Loads component at file path.

**Parameters** **file_path** (*str*) – Location to load file.

**Returns** ComponentBase object

**property name**(*cls*)

Returns string name of this component.

**needs_fitting**(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need to be fit or whose fit methods do nothing.

**Returns** True.

**property parameters**(*self*)

Returns the parameters which were used to initialize the component.

**save**(*self*, *file_path*, *pickle_protocol=cloudpickle.DEFAULT_PROTOCOL*)

Saves component at file path.

**Parameters**

- **file_path** (*str*) – Location to save file.
- **pickle_protocol** (*int*) – The pickle data stream format.

**transform**(*self*, *X*, *y=None*)

Transforms input data by selecting features. If the component_obj does not have a transform method, will raise an MethodPropertyNotFoundError exception.

**Parameters**

- **X** (*pd.DataFrame*) – Data to transform.
- **y** (*pd.Series, optional*) – Target data. Ignored.

**Returns** Transformed X

**Return type** pd.DataFrame

**Raises** **MethodPropertyNotFoundError** – If feature selector does not have a transform method or a component_obj that implements transform

**update_parameters**(*self*, *update_dict*, *reset_fit=True*)

Updates the parameter dictionary of the component.

**Parameters**

- **update_dict** (*dict*) – A dict of parameters to update.
- **reset_fit** (*bool, optional*) – If True, will set *_is_fitted* to False.

**class** evalml.pipelines.components.transformers.feature_selection.**RFClassifierRFESelector**(*step=0.2*, *min_features_to_se*
*cv=None*, *scor-*
*ing=None*, *n_jobs=-*

*1*, *n_estimators=10*, *max_depth=None*, *ran-*
*dom_seed=0*, ***kwargs*)

Selects relevant features using recursive feature elimination with a Random Forest Classifier.

**Parameters**

- **step** (*int, float*) – The number of features to eliminate in each iteration. If an integer is specified this will represent the number of features to eliminate. If a float is specified this represents the percentage of features to eliminate each iteration. The last iteration may drop fewer than this number of features in order to satisfy the min_features_to_select constraint. Defaults to 0.2.
- **min_features_to_select** (*int*) – The minimum number of features to return. Defaults to 1.
- **cv** (*int or None*) – Number of folds to use for the cross-validation splitting strategy. Defaults to None which will use 5 folds.
- **scoring** (*str, callable or None*) – A string or scorer callable object to specify the scoring method.
- **n_jobs** (*int or None*) – Number of jobs to run in parallel. -1 uses all processes. Defaults to -1.
- **n_estimators** (*int*) – The number of trees in the forest. Defaults to 10.
- **max_depth** (*int*) – Maximum tree depth for base learners. Defaults to None.
- **random_seed** (*int*) – Seed for the random number generator. Defaults to 0.

**Attributes**

| | |
|---|---|
| **hyper-parameter_ranges** | { "step": Real(0.05, 0.25)} |
| **modifies_features** | True |
| **modifies_target** | False |
| **name** | RFE Selector with RF Classifier |
| **training_only** | False |

**Methods**

| | |
|---|---|
| *clone* | Constructs a new component with the same parameters and random state. |
| *default_parameters* | Returns the default parameters for this component. |
| *describe* | Describe a component and its parameters. |
| *fit* | Fits component to data. |
| *fit_transform* | Fit and transform data using the feature selector. |
| *get_names* | Get names of selected features. |
| *load* | Loads component at file path. |
| *needs_fitting* | Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances. |
| *parameters* | Returns the parameters which were used to initialize the component. |
| *save* | Saves component at file path. |
| *transform* | Transforms input data by selecting features. If the component_obj does not have a transform method, will raise an MethodPropertyNotFoundError exception. |
| *update_parameters* | Updates the parameter dictionary of the component. |

**clone**(*self*)

Constructs a new component with the same parameters and random state.

**Returns** A new instance of this component with identical parameters and random state.

**default_parameters**(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

**Returns** Default parameters for this component.

**Return type** dict

**describe**(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

**Parameters**

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description as dictionary in the format {"name": name, "parameters": parameters}

**Returns** Returns dictionary if return_dict is True, else None.

**Return type** None or dict

**fit**(*self*, *X*, *y=None*)

Fits component to data.

**Parameters**

- **X** (*pd.DataFrame*) – The input training data of shape [n_samples, n_features]
- **y** (*pd.Series, optional*) – The target training data of length [n_samples]

**Returns** self

**Raises** `MethodPropertyNotFoundError` – If component does not have a fit method or a component_obj that implements fit.

`fit_transform`(*self*, *X*, *y=None*)

Fit and transform data using the feature selector.

**Parameters**

- `X` (*pd.DataFrame*) – The input training data of shape [n_samples, n_features].
- `y` (*pd.Series, optional*) – The target training data of length [n_samples].

**Returns** Transformed data.

**Return type** pd.DataFrame

`get_names`(*self*)

Get names of selected features.

**Returns** List of the names of features selected.

**Return type** list[str]

`static load`(*file_path*)

Loads component at file path.

**Parameters** `file_path` (*str*) – Location to load file.

**Returns** ComponentBase object

`needs_fitting`(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need to be fit or whose fit methods do nothing.

**Returns** True.

`property parameters`(*self*)

Returns the parameters which were used to initialize the component.

`save`(*self*, *file_path*, *pickle_protocol=cloudpickle.DEFAULT_PROTOCOL*)

Saves component at file path.

**Parameters**

- `file_path` (*str*) – Location to save file.
- `pickle_protocol` (*int*) – The pickle data stream format.

`transform`(*self*, *X*, *y=None*)

Transforms input data by selecting features. If the component_obj does not have a transform method, will raise an MethodPropertyNotFoundError exception.

**Parameters**

- `X` (*pd.DataFrame*) – Data to transform.
- `y` (*pd.Series, optional*) – Target data. Ignored.

**Returns** Transformed X

**Return type** pd.DataFrame

**Raises** `MethodPropertyNotFoundError` – If feature selector does not have a transform method or a component_obj that implements transform

**update_parameters**(*self*, *update_dict*, *reset_fit=True*)

Updates the parameter dictionary of the component.

**Parameters**

- **update_dict** (*dict*) – A dict of parameters to update.
- **reset_fit** (*bool, optional*) – If True, will set *_is_fitted* to False.

**class** evalml.pipelines.components.transformers.feature_selection.**RFClassifierSelectFromModel**(*number_featu n_estimators= max_depth=N per- cent_features thresh- old='median' n_jobs=- 1, ran- dom_seed=0, **kwargs*)

Selects top features based on importance weights using a Random Forest classifier.

**Parameters**

- **number_features** (*int*) – The maximum number of features to select. If both percent_features and number_features are specified, take the greater number of features. Defaults to None.
- **n_estimators** (*int*) – The number of trees in the forest. Defaults to 10.
- **max_depth** (*int*) – Maximum tree depth for base learners. Defaults to None.
- **percent_features** (*float*) – Percentage of features to use. If both percent_features and number_features are specified, take the greater number of features. Defaults to 0.5.
- **threshold** (*string or float*) – The threshold value to use for feature selection. Features whose importance is greater or equal are kept while the others are discarded. If "median", then the threshold value is the median of the feature importances. A scaling factor (e.g., "1.25*mean") may also be used. Defaults to median.
- **n_jobs** (*int or None*) – Number of jobs to run in parallel. -1 uses all processes. Defaults to -1.
- **random_seed** (*int*) – Seed for the random number generator. Defaults to 0.

**Attributes**

| | |
|---|---|
| **hyper-parame-ter_ranges** | { "percent_features": Real(0.01, 1), "threshold": ["mean", "median"],} |
| **modi-fies_features** | True |
| **modi-fies_target** | False |
| **name** | RF Classifier Select From Model |
| **train-ing_only** | False |

**Methods**

| | |
|---|---|
| *clone* | Constructs a new component with the same parameters and random state. |
| *default_parameters* | Returns the default parameters for this component. |
| *describe* | Describe a component and its parameters. |
| *fit* | Fits component to data. |
| *fit_transform* | Fit and transform data using the feature selector. |
| *get_names* | Get names of selected features. |
| *load* | Loads component at file path. |
| *needs_fitting* | Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances. |
| *parameters* | Returns the parameters which were used to initialize the component. |
| *save* | Saves component at file path. |
| *transform* | Transforms input data by selecting features. If the component_obj does not have a transform method, will raise an MethodPropertyNotFoundError exception. |
| *update_parameters* | Updates the parameter dictionary of the component. |

**clone**(*self*)

Constructs a new component with the same parameters and random state.

**Returns** A new instance of this component with identical parameters and random state.

**default_parameters**(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

**Returns** Default parameters for this component.

**Return type** dict

**describe**(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

**Parameters**

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description as dictionary in the format {"name": name, "parameters": parameters}

**Returns** Returns dictionary if return_dict is True, else None.

**Return type** None or dict

**fit**(*self*, *X*, *y=None*)

Fits component to data.

**Parameters**

- **X** (*pd.DataFrame*) – The input training data of shape [n_samples, n_features]
- **y** (*pd.Series, optional*) – The target training data of length [n_samples]

**Returns** self

**Raises** `MethodPropertyNotFoundError` – If component does not have a fit method or a component_obj that implements fit.

`fit_transform`(*self*, *X*, *y=None*)

Fit and transform data using the feature selector.

**Parameters**

- `X` (*pd.DataFrame*) – The input training data of shape [n_samples, n_features].
- `y` (*pd.Series, optional*) – The target training data of length [n_samples].

**Returns** Transformed data.

**Return type** pd.DataFrame

`get_names`(*self*)

Get names of selected features.

**Returns** List of the names of features selected.

**Return type** list[str]

`static load`(*file_path*)

Loads component at file path.

**Parameters** `file_path` (*str*) – Location to load file.

**Returns** ComponentBase object

`needs_fitting`(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need to be fit or whose fit methods do nothing.

**Returns** True.

`property parameters`(*self*)

Returns the parameters which were used to initialize the component.

`save`(*self*, *file_path*, *pickle_protocol=cloudpickle.DEFAULT_PROTOCOL*)

Saves component at file path.

**Parameters**

- `file_path` (*str*) – Location to save file.
- `pickle_protocol` (*int*) – The pickle data stream format.

`transform`(*self*, *X*, *y=None*)

Transforms input data by selecting features. If the component_obj does not have a transform method, will raise an MethodPropertyNotFoundError exception.

**Parameters**

- `X` (*pd.DataFrame*) – Data to transform.
- `y` (*pd.Series, optional*) – Target data. Ignored.

**Returns** Transformed X

**Return type** pd.DataFrame

**Raises** `MethodPropertyNotFoundError` – If feature selector does not have a transform method or a component_obj that implements transform

**update_parameters**(*self*, *update_dict*, *reset_fit=True*)

Updates the parameter dictionary of the component.

**Parameters**

- **update_dict** (*dict*) – A dict of parameters to update.
- **reset_fit** (*bool, optional*) – If True, will set *_is_fitted* to False.

**class** evalml.pipelines.components.transformers.feature_selection.**RFRegressorRFESelector**(*step=0.2*, *min_features_to_sel cv=None*, *scor- ing=None*, *n_jobs=- 1*, *n_estimators=10*, *max_depth=None*, *ran- dom_seed=0*, ***kwargs*)

Selects relevant features using recursive feature elimination with a Random Forest Regressor.

**Parameters**

- **step** (*int, float*) – The number of features to eliminate in each iteration. If an integer is specified this will represent the number of features to eliminate. If a float is specified this represents the percentage of features to eliminate each iteration. The last iteration may drop fewer than this number of features in order to satisfy the min_features_to_select constraint. Defaults to 0.2.
- **min_features_to_select** (*int*) – The minimum number of features to return. Defaults to 1.
- **cv** (*int or None*) – Number of folds to use for the cross-validation splitting strategy. Defaults to None which will use 5 folds.
- **scoring** (*str, callable or None*) – A string or scorer callable object to specify the scoring method.
- **n_jobs** (*int or None*) – Number of jobs to run in parallel. -1 uses all processes. Defaults to -1.
- **n_estimators** (*int*) – The number of trees in the forest. Defaults to 10.
- **max_depth** (*int*) – Maximum tree depth for base learners. Defaults to None.
- **random_seed** (*int*) – Seed for the random number generator. Defaults to 0.

**Attributes**

| | |
|---|---|
| **hyper-parameter_ranges** | { "step": Real(0.05, 0.25)} |
| **modifies_features** | True |
| **modifies_target** | False |
| **name** | RFE Selector with RF Regressor |
| **training_only** | False |

**Methods**

| | |
|---|---|
| *clone* | Constructs a new component with the same parameters and random state. |
| *default_parameters* | Returns the default parameters for this component. |
| *describe* | Describe a component and its parameters. |
| *fit* | Fits component to data. |
| *fit_transform* | Fit and transform data using the feature selector. |
| *get_names* | Get names of selected features. |
| *load* | Loads component at file path. |
| *needs_fitting* | Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances. |
| *parameters* | Returns the parameters which were used to initialize the component. |
| *save* | Saves component at file path. |
| *transform* | Transforms input data by selecting features. If the component_obj does not have a transform method, will raise an MethodPropertyNotFoundError exception. |
| *update_parameters* | Updates the parameter dictionary of the component. |

**clone**(*self*)

Constructs a new component with the same parameters and random state.

**Returns** A new instance of this component with identical parameters and random state.

**default_parameters**(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

**Returns** Default parameters for this component.

**Return type** dict

**describe**(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

**Parameters**

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description as dictionary in the format {"name": name, "parameters": parameters}

**Returns** Returns dictionary if return_dict is True, else None.

**Return type** None or dict

`fit`(*self*, *X*, *y=None*)

Fits component to data.

**Parameters**

- **X** (*pd.DataFrame*) – The input training data of shape [n_samples, n_features]
- **y** (*pd.Series, optional*) – The target training data of length [n_samples]

**Returns** self

**Raises** **MethodPropertyNotFoundError** – If component does not have a fit method or a component_obj that implements fit.

`fit_transform`(*self*, *X*, *y=None*)

Fit and transform data using the feature selector.

**Parameters**

- **X** (*pd.DataFrame*) – The input training data of shape [n_samples, n_features].
- **y** (*pd.Series, optional*) – The target training data of length [n_samples].

**Returns** Transformed data.

**Return type** pd.DataFrame

`get_names`(*self*)

Get names of selected features.

**Returns** List of the names of features selected.

**Return type** list[str]

`static load`(*file_path*)

Loads component at file path.

**Parameters** **file_path** (*str*) – Location to load file.

**Returns** ComponentBase object

`needs_fitting`(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need to be fit or whose fit methods do nothing.

**Returns** True.

`property parameters`(*self*)

Returns the parameters which were used to initialize the component.

`save`(*self*, *file_path*, *pickle_protocol=cloudpickle.DEFAULT_PROTOCOL*)

Saves component at file path.

**Parameters**

- **file_path** (*str*) – Location to save file.
- **pickle_protocol** (*int*) – The pickle data stream format.

**transform**(*self*, *X*, *y=None*)

Transforms input data by selecting features. If the component_obj does not have a transform method, will raise an MethodPropertyNotFoundError exception.

**Parameters**

- **X** (*pd.DataFrame*) – Data to transform.
- **y** (*pd.Series, optional*) – Target data. Ignored.

**Returns** Transformed X

**Return type** pd.DataFrame

**Raises** **MethodPropertyNotFoundError** – If feature selector does not have a transform method or a component_obj that implements transform

**update_parameters**(*self*, *update_dict*, *reset_fit=True*)

Updates the parameter dictionary of the component.

**Parameters**

- **update_dict** (*dict*) – A dict of parameters to update.
- **reset_fit** (*bool, optional*) – If True, will set *_is_fitted* to False.

**class** evalml.pipelines.components.transformers.feature_selection.**RFRegressorSelectFromModel**(*number_featur n_estimators= max_depth=N per- cent_features= thresh- old='median', n_jobs=- 1, ran- dom_seed=0, **kwargs*)

Selects top features based on importance weights using a Random Forest regressor.

**Parameters**

- **number_features** (*int*) – The maximum number of features to select. If both percent_features and number_features are specified, take the greater number of features. Defaults to 0.5.
- **n_estimators** (*int*) – The number of trees in the forest. Defaults to 10.
- **max_depth** (*int*) – Maximum tree depth for base learners. Defaults to None.
- **percent_features** (*float*) – Percentage of features to use. If both percent_features and number_features are specified, take the greater number of features. Defaults to 0.5.
- **threshold** (*string or float*) – The threshold value to use for feature selection. Features whose importance is greater or equal are kept while the others are discarded. If "median", then the threshold value is the median of the feature importances. A scaling factor (e.g., "1.25*mean") may also be used. Defaults to median.
- **n_jobs** (*int or None*) – Number of jobs to run in parallel. -1 uses all processes. Defaults to -1.
- **random_seed** (*int*) – Seed for the random number generator. Defaults to 0.

**Attributes**

| | |
|---|---|
| **hyper-parameter_ranges** | { "percent_features": Real(0.01, 1), "threshold": ["mean", "median"],} |
| **modifies_features** | True |
| **modifies_target** | False |
| **name** | RF Regressor Select From Model |
| **training_only** | False |

**Methods**

| | |
|---|---|
| *clone* | Constructs a new component with the same parameters and random state. |
| *default_parameters* | Returns the default parameters for this component. |
| *describe* | Describe a component and its parameters. |
| *fit* | Fits component to data. |
| *fit_transform* | Fit and transform data using the feature selector. |
| *get_names* | Get names of selected features. |
| *load* | Loads component at file path. |
| *needs_fitting* | Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances. |
| *parameters* | Returns the parameters which were used to initialize the component. |
| *save* | Saves component at file path. |
| *transform* | Transforms input data by selecting features. If the component_obj does not have a transform method, will raise an MethodPropertyNotFoundError exception. |
| *update_parameters* | Updates the parameter dictionary of the component. |

**clone**(*self*)

Constructs a new component with the same parameters and random state.

**Returns** A new instance of this component with identical parameters and random state.

**default_parameters**(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

**Returns** Default parameters for this component.

**Return type** dict

**describe**(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

**Parameters**

- **print_name** (*bool, optional*) – whether to print name of component

- **return_dict** (*bool, optional*) – whether to return description as dictionary in the format {"name": name, "parameters": parameters}

**Returns** Returns dictionary if return_dict is True, else None.

**Return type** None or dict

**fit**(*self*, *X*, *y=None*)

Fits component to data.

**Parameters**

- **X** (*pd.DataFrame*) – The input training data of shape [n_samples, n_features]
- **y** (*pd.Series, optional*) – The target training data of length [n_samples]

**Returns** self

**Raises** **MethodPropertyNotFoundError** – If component does not have a fit method or a component_obj that implements fit.

**fit_transform**(*self*, *X*, *y=None*)

Fit and transform data using the feature selector.

**Parameters**

- **X** (*pd.DataFrame*) – The input training data of shape [n_samples, n_features].
- **y** (*pd.Series, optional*) – The target training data of length [n_samples].

**Returns** Transformed data.

**Return type** pd.DataFrame

**get_names**(*self*)

Get names of selected features.

**Returns** List of the names of features selected.

**Return type** list[str]

**static load**(*file_path*)

Loads component at file path.

**Parameters** **file_path** (*str*) – Location to load file.

**Returns** ComponentBase object

**needs_fitting**(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need to be fit or whose fit methods do nothing.

**Returns** True.

**property parameters**(*self*)

Returns the parameters which were used to initialize the component.

**save**(*self*, *file_path*, *pickle_protocol=cloudpickle.DEFAULT_PROTOCOL*)

Saves component at file path.

**Parameters**

- **file_path** (*str*) – Location to save file.

- **pickle_protocol** (*int*) – The pickle data stream format.

**transform**(*self*, *X*, *y=None*)

Transforms input data by selecting features. If the component_obj does not have a transform method, will raise an MethodPropertyNotFoundError exception.

**Parameters**

- **X** (*pd.DataFrame*) – Data to transform.
- **y** (*pd.Series, optional*) – Target data. Ignored.

**Returns** Transformed X

**Return type** pd.DataFrame

**Raises** **MethodPropertyNotFoundError** – If feature selector does not have a transform method or a component_obj that implements transform

**update_parameters**(*self*, *update_dict*, *reset_fit=True*)

Updates the parameter dictionary of the component.

**Parameters**

- **update_dict** (*dict*) – A dict of parameters to update.
- **reset_fit** (*bool, optional*) – If True, will set *_is_fitted* to False.

## imputers

Components that impute missing values in the input data.

### Submodules

### imputer

Component that imputes missing data according to a specified imputation strategy.

### Module Contents

### Classes Summary

| | |
|---|---|
| *Imputer* | Imputes missing data according to a specified imputation strategy. |

## Contents

**class** evalml.pipelines.components.transformers.imputers.imputer.**Imputer**(*categorical_impute_strategy='most_freq*
*categori-*
*cal_fill_value=None*,
*nu-*
*meric_impute_strategy='mean'*,
*nu-*
*meric_fill_value=None*,
*boolean_impute_strategy='most_frequen*
*boolean_fill_value=None*,
*random_seed=0*,
***kwargs*)

Imputes missing data according to a specified imputation strategy.

**Parameters**

- **categorical_impute_strategy** (*string*) – Impute strategy to use for string, object, boolean, categorical dtypes. Valid values include "most_frequent" and "constant".
- **numeric_impute_strategy** (*string*) – Impute strategy to use for numeric columns. Valid values include "mean", "median", "most_frequent", and "constant".
- **boolean_impute_strategy** (*string*) – Impute strategy to use for boolean columns. Valid values include "most_frequent" and "constant".
- **categorical_fill_value** (*string*) – When categorical_impute_strategy == "constant", fill_value is used to replace missing data. The default value of None will fill with the string "missing_value".
- **numeric_fill_value** (*int, float*) – When numeric_impute_strategy == "constant", fill_value is used to replace missing data. The default value of None will fill with 0.
- **boolean_fill_value** (*bool*) – When boolean_impute_strategy == "constant", fill_value is used to replace missing data. The default value of None will fill with True.
- **random_seed** (*int*) – Seed for the random number generator. Defaults to 0.

**Attributes**

| | |
|---|---|
| **hyper-parame-ter_ranges** | { "categorical_impute_strategy": ["most_frequent"], "numeric_impute_strategy": ["mean", "median", "most_frequent", "knn"], "boolean_impute_strategy": ["most_frequent"]} |
| **modi-fies_features** | True |
| **modi-fies_target** | False |
| **name** | Imputer |
| **train-ing_only** | False |

**Methods**

| | |
|---|---|
| *clone* | Constructs a new component with the same parameters and random state. |
| *default_parameters* | Returns the default parameters for this component. |
| *describe* | Describe a component and its parameters. |
| *fit* | Fits imputer to data. 'None' values are converted to np.nan before imputation and are treated as the same. |
| *fit_transform* | Fits on X and transforms X. |
| *load* | Loads component at file path. |
| *needs_fitting* | Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances. |
| *parameters* | Returns the parameters which were used to initialize the component. |
| *save* | Saves component at file path. |
| *transform* | Transforms data X by imputing missing values. |
| *update_parameters* | Updates the parameter dictionary of the component. |

**clone**(*self*)

Constructs a new component with the same parameters and random state.

**Returns** A new instance of this component with identical parameters and random state.

**default_parameters**(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

**Returns** Default parameters for this component.

**Return type** dict

**describe**(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

**Parameters**

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description as dictionary in the format {"name": name, "parameters": parameters}

**Returns** Returns dictionary if return_dict is True, else None.

**Return type** None or dict

**fit**(*self*, *X*, *y=None*)

Fits imputer to data. 'None' values are converted to np.nan before imputation and are treated as the same.

**Parameters**

- **X** (*pd.DataFrame, np.ndarray*) – The input training data of shape [n_samples, n_features]
- **y** (*pd.Series, optional*) – The target training data of length [n_samples]

**Returns** self

**fit_transform**(*self*, *X*, *y=None*)

Fits on X and transforms X.

**Parameters**

- **X** (*pd.DataFrame*) – Data to fit and transform.
- **y** (*pd.Series*) – Target data.

**Returns** Transformed X.

**Return type** pd.DataFrame

**Raises** **MethodPropertyNotFoundError** – If transformer does not have a transform method or a component_obj that implements transform.

**static load**(*file_path*)

Loads component at file path.

**Parameters** **file_path** (*str*) – Location to load file.

**Returns** ComponentBase object

**needs_fitting**(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need to be fit or whose fit methods do nothing.

**Returns** True.

**property parameters**(*self*)

Returns the parameters which were used to initialize the component.

**save**(*self*, *file_path*, *pickle_protocol=cloudpickle.DEFAULT_PROTOCOL*)

Saves component at file path.

**Parameters**

- **file_path** (*str*) – Location to save file.
- **pickle_protocol** (*int*) – The pickle data stream format.

**transform**(*self*, *X*, *y=None*)

Transforms data X by imputing missing values.

**Parameters**

- **X** (*pd.DataFrame*) – Data to transform
- **y** (*pd.Series, optional*) – Ignored.

**Returns** Transformed X

**Return type** pd.DataFrame

**update_parameters**(*self*, *update_dict*, *reset_fit=True*)

Updates the parameter dictionary of the component.

**Parameters**

- **update_dict** (*dict*) – A dict of parameters to update.
- **reset_fit** (*bool, optional*) – If True, will set *_is_fitted* to False.

### knn_imputer

Component that imputes missing data according to a specified imputation strategy.

### Module Contents

### Classes Summary

| | |
|---|---|
| *KNNImputer* | Imputes missing data using KNN according to a specified number of neighbors. Natural language columns are ignored. |

### Contents

**class** evalml.pipelines.components.transformers.imputers.knn_imputer.**KNNImputer**(*number_neighbors=3, random_seed=0, **kwargs*)

Imputes missing data using KNN according to a specified number of neighbors. Natural language columns are ignored.

**Parameters**

- **number_neighbors** (*int*) – Number of nearest neighbors for KNN to search for. Defaults to 3.
- **random_seed** (*int*) – Seed for the random number generator. Defaults to 0.

**Attributes**

| | |
|---|---|
| **modifies_features** | True |
| **modifies_target** | False |
| **name** | KNN Imputer |
| **training_only** | False |

**Methods**

| | |
|---|---|
| *clone* | Constructs a new component with the same parameters and random state. |
| *default_parameters* | Returns the default parameters for this component. |
| *describe* | Describe a component and its parameters. |
| *fit* | Fits imputer to data. 'None' values are converted to np.nan before imputation and are treated as the same. |
| *fit_transform* | Fits on X and transforms X. |
| *load* | Loads component at file path. |
| *needs_fitting* | Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances. |
| *parameters* | Returns the parameters which were used to initialize the component. |
| *save* | Saves component at file path. |
| *transform* | Transforms input by imputing missing values. 'None' and np.nan values are treated as the same. |
| *update_parameters* | Updates the parameter dictionary of the component. |

**clone**(*self*)

Constructs a new component with the same parameters and random state.

**Returns** A new instance of this component with identical parameters and random state.

**default_parameters**(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

**Returns** Default parameters for this component.

**Return type** dict

**describe**(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

**Parameters**

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description as dictionary in the format {"name": name, "parameters": parameters}

**Returns** Returns dictionary if return_dict is True, else None.

**Return type** None or dict

**fit**(*self*, *X*, *y=None*)

Fits imputer to data. 'None' values are converted to np.nan before imputation and are treated as the same.

**Parameters**

- **X** (*pd.DataFrame or np.ndarray*) – the input training data of shape [n_samples, n_features]
- **y** (*pd.Series, optional*) – the target training data of length [n_samples]

**Returns** self

**Raises** **ValueError** – if the KNNImputer receives a dataframe with both Boolean and Categorical data.

**fit_transform**(*self*, *X*, *y=None*)

Fits on X and transforms X.

**Parameters**

- **X** (*pd.DataFrame*) – Data to fit and transform
- **y** (*pd.Series, optional*) – Target data.

**Returns** Transformed X

**Return type** pd.DataFrame

**static load**(*file_path*)

Loads component at file path.

**Parameters** **file_path** (*str*) – Location to load file.

**Returns** ComponentBase object

**needs_fitting**(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need to be fit or whose fit methods do nothing.

**Returns** True.

**property parameters**(*self*)

Returns the parameters which were used to initialize the component.

**save**(*self*, *file_path*, *pickle_protocol=cloudpickle.DEFAULT_PROTOCOL*)

Saves component at file path.

**Parameters**

- **file_path** (*str*) – Location to save file.
- **pickle_protocol** (*int*) – The pickle data stream format.

**transform**(*self*, *X*, *y=None*)

Transforms input by imputing missing values. 'None' and np.nan values are treated as the same.

**Parameters**

- **X** (*pd.DataFrame*) – Data to transform.
- **y** (*pd.Series, optional*) – Ignored.

**Returns** Transformed X

**Return type** pd.DataFrame

**update_parameters**(*self*, *update_dict*, *reset_fit=True*)

Updates the parameter dictionary of the component.

**Parameters**

- **update_dict** (*dict*) – A dict of parameters to update.
- **reset_fit** (*bool, optional*) – If True, will set *_is_fitted* to False.

### per_column_imputer

Component that imputes missing data according to a specified imputation strategy per column.

### Module Contents

### Classes Summary

| | |
|---|---|
| *PerColumnImputer* | Imputes missing data according to a specified imputation strategy per column. |

### Contents

**class** evalml.pipelines.components.transformers.imputers.per_column_imputer.**PerColumnImputer**(*impute_strateg ran-dom_seed=0, **kwargs*)

Imputes missing data according to a specified imputation strategy per column.

**Parameters**

- **impute_strategies** (*dict*) – Column and {"impute_strategy": strategy, "fill_value":value} pairings. Valid values for impute strategy include "mean", "median", "most_frequent", "constant" for numerical data, and "most_frequent", "constant" for object data types. Defaults to None, which uses "most_frequent" for all columns. When impute_strategy == "constant", fill_value is used to replace missing data. When None, uses 0 when imputing numerical data and "missing_value" for strings or object data types.
- **random_seed** (*int*) – Seed for the random number generator. Defaults to 0.

**Attributes**

| | |
|---|---|
| **hyper-parame-ter_ranges** | {} |
| **modi-fies_features** | True |
| **modi-fies_target** | False |
| **name** | Per Column Imputer |
| **train-ing_only** | False |

**Methods**

| | |
|---|---|
| *clone* | Constructs a new component with the same parameters and random state. |
| *default_parameters* | Returns the default parameters for this component. |
| *describe* | Describe a component and its parameters. |
| *fit* | Fits imputers on input data. |
| *fit_transform* | Fits on X and transforms X. |
| *load* | Loads component at file path. |
| *needs_fitting* | Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances. |
| *parameters* | Returns the parameters which were used to initialize the component. |
| *save* | Saves component at file path. |
| *transform* | Transforms input data by imputing missing values. |
| *update_parameters* | Updates the parameter dictionary of the component. |

**clone**(*self*)

Constructs a new component with the same parameters and random state.

**Returns** A new instance of this component with identical parameters and random state.

**default_parameters**(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

**Returns** Default parameters for this component.

**Return type** dict

**describe**(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

**Parameters**

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description as dictionary in the format {"name": name, "parameters": parameters}

**Returns** Returns dictionary if return_dict is True, else None.

**Return type** None or dict

**fit**(*self*, *X*, *y=None*)

Fits imputers on input data.

**Parameters**

- **X** (*pd.DataFrame or np.ndarray*) – The input training data of shape [n_samples, n_features] to fit.
- **y** (*pd.Series, optional*) – The target training data of length [n_samples]. Ignored.

**Returns** self

**fit_transform**(*self*, *X*, *y=None*)

Fits on X and transforms X.

**Parameters**

- **X** (*pd.DataFrame*) – Data to fit and transform.
- **y** (*pd.Series*) – Target data.

**Returns** Transformed X.

**Return type** pd.DataFrame

**Raises** **MethodPropertyNotFoundError** – If transformer does not have a transform method or a component_obj that implements transform.

**static load**(*file_path*)

Loads component at file path.

**Parameters** **file_path** (*str*) – Location to load file.

**Returns** ComponentBase object

**needs_fitting**(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need to be fit or whose fit methods do nothing.

**Returns** True.

**property parameters**(*self*)

Returns the parameters which were used to initialize the component.

**save**(*self*, *file_path*, *pickle_protocol=cloudpickle.DEFAULT_PROTOCOL*)

Saves component at file path.

**Parameters**

- **file_path** (*str*) – Location to save file.
- **pickle_protocol** (*int*) – The pickle data stream format.

**transform**(*self*, *X*, *y=None*)

Transforms input data by imputing missing values.

**Parameters**

- **X** (*pd.DataFrame or np.ndarray*) – The input training data of shape [n_samples, n_features] to transform.
- **y** (*pd.Series, optional*) – The target training data of length [n_samples]. Ignored.

**Returns** Transformed X

**Return type** pd.DataFrame

**update_parameters**(*self*, *update_dict*, *reset_fit=True*)

Updates the parameter dictionary of the component.

**Parameters**

- **update_dict** (*dict*) – A dict of parameters to update.
- **reset_fit** (*bool, optional*) – If True, will set *_is_fitted* to False.

### simple_imputer

Component that imputes missing data according to a specified imputation strategy.

### Module Contents

### Classes Summary

| | |
|---|---|
| *SimpleImputer* | Imputes missing data according to a specified imputation strategy. Natural language columns are ignored. |

### Contents

**class** evalml.pipelines.components.transformers.imputers.simple_imputer.**SimpleImputer**(*impute_strategy='most_fill_value=None, random_seed=0, **kwargs*)

Imputes missing data according to a specified imputation strategy. Natural language columns are ignored.

**Parameters**

- **impute_strategy** (*string*) – Impute strategy to use. Valid values include "mean", "median", "most_frequent", "constant" for numerical data, and "most_frequent", "constant" for object data types.
- **fill_value** (*string*) – When impute_strategy == "constant", fill_value is used to replace missing data. Defaults to 0 when imputing numerical data and "missing_value" for strings or object data types.
- **random_seed** (*int*) – Seed for the random number generator. Defaults to 0.

**Attributes**

| | |
|---|---|
| **hyperparameter_ranges** | { "impute_strategy": ["mean", "median", "most_frequent"]} |
| **modifies_features** | True |
| **modifies_target** | False |
| **name** | Simple Imputer |
| **training_only** | False |

**Methods**

| | |
|---|---|
| *clone* | Constructs a new component with the same parameters and random state. |
| *default_parameters* | Returns the default parameters for this component. |
| *describe* | Describe a component and its parameters. |
| *fit* | Fits imputer to data. 'None' values are converted to np.nan before imputation and are treated as the same. |
| *fit_transform* | Fits on X and transforms X. |
| *load* | Loads component at file path. |
| *needs_fitting* | Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances. |
| *parameters* | Returns the parameters which were used to initialize the component. |
| *save* | Saves component at file path. |
| *transform* | Transforms input by imputing missing values. 'None' and np.nan values are treated as the same. |
| *update_parameters* | Updates the parameter dictionary of the component. |

**clone**(*self*)

Constructs a new component with the same parameters and random state.

**Returns** A new instance of this component with identical parameters and random state.

**default_parameters**(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

**Returns** Default parameters for this component.

**Return type** dict

**describe**(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

**Parameters**

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description as dictionary in the format {"name": name, "parameters": parameters}

**Returns** Returns dictionary if return_dict is True, else None.

**Return type** None or dict

**fit**(*self*, *X*, *y=None*)

Fits imputer to data. 'None' values are converted to np.nan before imputation and are treated as the same.

**Parameters**

- **X** (*pd.DataFrame or np.ndarray*) – the input training data of shape [n_samples, n_features]
- **y** (*pd.Series, optional*) – the target training data of length [n_samples]

**Returns** self

**Raises** **ValueError** – if the SimpleImputer receives a dataframe with both Boolean and Categorical data.

**fit_transform**(*self*, *X*, *y=None*)

Fits on X and transforms X.

**Parameters**

- **X** (*pd.DataFrame*) – Data to fit and transform
- **y** (*pd.Series, optional*) – Target data.

**Returns** Transformed X

**Return type** pd.DataFrame

**static load**(*file_path*)

Loads component at file path.

**Parameters** **file_path** (*str*) – Location to load file.

**Returns** ComponentBase object

**needs_fitting**(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need to be fit or whose fit methods do nothing.

**Returns** True.

**property parameters**(*self*)

Returns the parameters which were used to initialize the component.

**save**(*self*, *file_path*, *pickle_protocol=cloudpickle.DEFAULT_PROTOCOL*)

Saves component at file path.

**Parameters**

- **file_path** (*str*) – Location to save file.
- **pickle_protocol** (*int*) – The pickle data stream format.

**transform**(*self*, *X*, *y=None*)

Transforms input by imputing missing values. 'None' and np.nan values are treated as the same.

**Parameters**

- **X** (*pd.DataFrame*) – Data to transform.
- **y** (*pd.Series, optional*) – Ignored.

**Returns** Transformed X

**Return type** pd.DataFrame

**update_parameters**(*self*, *update_dict*, *reset_fit=True*)

Updates the parameter dictionary of the component.

**Parameters**

- **update_dict** (*dict*) – A dict of parameters to update.
- **reset_fit** (*bool, optional*) – If True, will set *_is_fitted* to False.

### target_imputer

Component that imputes missing target data according to a specified imputation strategy.

### Module Contents

### Classes Summary

| | |
|---|---|
| *TargetImputer* | Imputes missing target data according to a specified imputation strategy. |
| *TargetImputerMeta* | A version of the ComponentBaseMeta class which handles when input features is None. |

### Contents

**class** evalml.pipelines.components.transformers.imputers.target_imputer.**TargetImputer**(*impute_strategy='most_ fill_value=None, random_seed=0, **kwargs*)

Imputes missing target data according to a specified imputation strategy.

**Parameters**

- **impute_strategy** (*string*) – Impute strategy to use. Valid values include "mean", "median", "most_frequent", "constant" for numerical data, and "most_frequent", "constant" for object data types. Defaults to "most_frequent".
- **fill_value** (*string*) – When impute_strategy == "constant", fill_value is used to replace missing data. Defaults to None which uses 0 when imputing numerical data and "missing_value" for strings or object data types.
- **random_seed** (*int*) – Seed for the random number generator. Defaults to 0.

**Attributes**

| | |
|---|---|
| **hyperparameter_ranges** | { "impute_strategy": ["mean", "median", "most_frequent"]} |
| **modifies_features** | False |
| **modifies_target** | True |
| **name** | Target Imputer |
| **training_only** | False |

**Methods**

| | |
|---|---|
| *clone* | Constructs a new component with the same parameters and random state. |
| *default_parameters* | Returns the default parameters for this component. |
| *describe* | Describe a component and its parameters. |
| *fit* | Fits imputer to target data. 'None' values are converted to np.nan before imputation and are treated as the same. |
| *fit_transform* | Fits on and transforms the input target data. |
| *load* | Loads component at file path. |
| *needs_fitting* | Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances. |
| *parameters* | Returns the parameters which were used to initialize the component. |
| *save* | Saves component at file path. |
| *transform* | Transforms input target data by imputing missing values. 'None' and np.nan values are treated as the same. |
| *update_parameters* | Updates the parameter dictionary of the component. |

**clone**(*self*)

Constructs a new component with the same parameters and random state.

**Returns** A new instance of this component with identical parameters and random state.

**default_parameters**(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

**Returns** Default parameters for this component.

**Return type** dict

**describe**(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

**Parameters**

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description as dictionary in the format {"name": name, "parameters": parameters}

**Returns** Returns dictionary if return_dict is True, else None.

**Return type** None or dict

**fit**(*self*, *X*, *y*)

Fits imputer to target data. 'None' values are converted to np.nan before imputation and are treated as the same.

**Parameters**

- **X** (*pd.DataFrame or np.ndarray*) – The input training data of shape [n_samples, n_features]. Ignored.
- **y** (*pd.Series, optional*) – The target training data of length [n_samples].

**Returns** self

**Raises** `TypeError` – If target is filled with all null values.

`fit_transform`(*self*, *X*, *y*)

Fits on and transforms the input target data.

**Parameters**

- `X` (*pd.DataFrame*) – Features. Ignored.
- `y` (*pd.Series*) – Target data to impute.

**Returns** The original X, transformed y

**Return type** (pd.DataFrame, pd.Series)

`static load`(*file_path*)

Loads component at file path.

**Parameters** `file_path` (*str*) – Location to load file.

**Returns** ComponentBase object

`needs_fitting`(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need to be fit or whose fit methods do nothing.

**Returns** True.

`property parameters`(*self*)

Returns the parameters which were used to initialize the component.

`save`(*self*, *file_path*, *pickle_protocol=cloudpickle.DEFAULT_PROTOCOL*)

Saves component at file path.

**Parameters**

- `file_path` (*str*) – Location to save file.
- `pickle_protocol` (*int*) – The pickle data stream format.

`transform`(*self*, *X*, *y*)

Transforms input target data by imputing missing values. 'None' and np.nan values are treated as the same.

**Parameters**

- `X` (*pd.DataFrame*) – Features. Ignored.
- `y` (*pd.Series*) – Target data to impute.

**Returns** The original X, transformed y

**Return type** (pd.DataFrame, pd.Series)

`update_parameters`(*self*, *update_dict*, *reset_fit=True*)

Updates the parameter dictionary of the component.

**Parameters**

- `update_dict` (*dict*) – A dict of parameters to update.
- `reset_fit` (*bool, optional*) – If True, will set *_is_fitted* to False.

**class** evalml.pipelines.components.transformers.imputers.target_imputer.**TargetImputerMeta**

A version of the ComponentBaseMeta class which handles when input features is None.

**Attributes**

| | |
|---|---|
| **FIT_METHODS** | ['fit', 'fit_transform'] |
| **METHODS_TO_CHECK** | ['predict', 'predict_proba', 'transform', 'inverse_transform', 'get_trend_dataframe'] |
| **PROPERTIES_TO_CHECK** | ['feature_importance'] |

**Methods**

| | |
|---|---|
| *check_for_fit* | *check_for_fit* wraps a method that validates if *self._is_fitted* is *True*. |
| *register* | Register a virtual subclass of an ABC. |
| *set_fit* | Wrapper for the fit method. |

**classmethod check_for_fit**(*cls*, *method*)

*check_for_fit* wraps a method that validates if *self._is_fitted* is *True*.

**Parameters** **method** (*callable*) – Method to wrap.

**Raises** **ComponentNotYetFittedError** – If component is not fitted.

**Returns** The wrapped input method.

**register**(*cls*, *subclass*)

Register a virtual subclass of an ABC.

Returns the subclass, to allow usage as a class decorator.

**classmethod set_fit**(*cls*, *method*)

Wrapper for the fit method.

## time_series_imputer

Component that imputes missing data according to a specified timeseries-specific imputation strategy.

### Module Contents

### Classes Summary

| | |
|---|---|
| *TimeSeriesImputer* | Imputes missing data according to a specified timeseries-specific imputation strategy. |

## Contents

**class** evalml.pipelines.components.transformers.imputers.time_series_imputer.**TimeSeriesImputer**(*categorical_nu-meric_impu tar-get_impute_ ran-dom_seed=( **kwargs*)

Imputes missing data according to a specified timeseries-specific imputation strategy.

This Transformer should be used after the *TimeSeriesRegularizer* in order to impute the missing values that were added to X and y (if passed).

**Parameters**

- **categorical_impute_strategy** (*string*) – Impute strategy to use for string, object, boolean, categorical dtypes. Valid values include "backwards_fill" and "forwards_fill". Defaults to "forwards_fill".
- **numeric_impute_strategy** (*string*) – Impute strategy to use for numeric columns. Valid values include "backwards_fill", "forwards_fill", and "interpolate". Defaults to "interpolate".
- **target_impute_strategy** (*string*) – Impute strategy to use for the target column. Valid values include "backwards_fill", "forwards_fill", and "interpolate". Defaults to "forwards_fill".
- **random_seed** (*int*) – Seed for the random number generator. Defaults to 0.

**Raises** **ValueError** – If categorical_impute_strategy, numeric_impute_strategy, or target_impute_strategy is not one of the valid values.

**Attributes**

| | |
|---|---|
| **hyper-parame-ter_ranges** | { "categorical_impute_strategy": ["backwards_fill", "forwards_fill"], "numeric_impute_strategy": ["backwards_fill", "forwards_fill", "interpolate"], "target_impute_strategy": ["backwards_fill", "forwards_fill", "interpolate"],} |
| **modi-fies_features** | True |
| **modi-fies_target** | True |
| **name** | Time Series Imputer |
| **train-ing_only** | True |

**Methods**

| | |
|---|---|
| *clone* | Constructs a new component with the same parameters and random state. |
| *default_parameters* | Returns the default parameters for this component. |
| *describe* | Describe a component and its parameters. |
| *fit* | Fits imputer to data. |
| *fit_transform* | Fits on X and transforms X. |
| *load* | Loads component at file path. |
| *needs_fitting* | Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances. |
| *parameters* | Returns the parameters which were used to initialize the component. |
| *save* | Saves component at file path. |
| *transform* | Transforms data X by imputing missing values using specified timeseries-specific strategies. 'None' values are converted to np.nan before imputation and are treated as the same. |
| *update_parameters* | Updates the parameter dictionary of the component. |

**clone**(*self*)

Constructs a new component with the same parameters and random state.

**Returns** A new instance of this component with identical parameters and random state.

**default_parameters**(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

**Returns** Default parameters for this component.

**Return type** dict

**describe**(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

**Parameters**

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description as dictionary in the format {"name": name, "parameters": parameters}

**Returns** Returns dictionary if return_dict is True, else None.

**Return type** None or dict

**fit**(*self*, *X*, *y=None*)

Fits imputer to data.

'None' values are converted to np.nan before imputation and are treated as the same. If a value is missing at the beginning or end of a column, that value will be imputed using backwards fill or forwards fill as necessary, respectively.

**Parameters**

- **X** (*pd.DataFrame, np.ndarray*) – The input training data of shape [n_samples, n_features]
- **y** (*pd.Series, optional*) – The target training data of length [n_samples]

**Returns** self

**fit_transform**(*self*, *X*, *y=None*)

Fits on X and transforms X.

**Parameters**

- **X** (*pd.DataFrame*) – Data to fit and transform.
- **y** (*pd.Series*) – Target data.

**Returns** Transformed X.

**Return type** pd.DataFrame

**Raises** **MethodPropertyNotFoundError** – If transformer does not have a transform method or a component_obj that implements transform.

**static load**(*file_path*)

Loads component at file path.

**Parameters** **file_path** (*str*) – Location to load file.

**Returns** ComponentBase object

**needs_fitting**(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need to be fit or whose fit methods do nothing.

**Returns** True.

**property parameters**(*self*)

Returns the parameters which were used to initialize the component.

**save**(*self*, *file_path*, *pickle_protocol=cloudpickle.DEFAULT_PROTOCOL*)

Saves component at file path.

**Parameters**

- **file_path** (*str*) – Location to save file.
- **pickle_protocol** (*int*) – The pickle data stream format.

**transform**(*self*, *X*, *y=None*)

Transforms data X by imputing missing values using specified timeseries-specific strategies. 'None' values are converted to np.nan before imputation and are treated as the same.

**Parameters**

- **X** (*pd.DataFrame*) – Data to transform.
- **y** (*pd.Series, optional*) – Optionally, target data to transform.

**Returns** Transformed X and y

**Return type** pd.DataFrame

**update_parameters**(*self*, *update_dict*, *reset_fit=True*)

Updates the parameter dictionary of the component.

**Parameters**

- **update_dict** (*dict*) – A dict of parameters to update.
- **reset_fit** (*bool, optional*) – If True, will set *_is_fitted* to False.

### Package Contents

### Classes Summary

| | |
|---|---|
| *Imputer* | Imputes missing data according to a specified imputation strategy. |
| *KNNImputer* | Imputes missing data using KNN according to a specified number of neighbors. Natural language columns are ignored. |
| *PerColumnImputer* | Imputes missing data according to a specified imputation strategy per column. |
| *SimpleImputer* | Imputes missing data according to a specified imputation strategy. Natural language columns are ignored. |
| *TargetImputer* | Imputes missing target data according to a specified imputation strategy. |
| *TimeSeriesImputer* | Imputes missing data according to a specified timeseries-specific imputation strategy. |

### Contents

**class** evalml.pipelines.components.transformers.imputers.**Imputer**(*categorical_impute_strategy='most_frequent', categorical_fill_value=None, numeric_impute_strategy='mean', numeric_fill_value=None, boolean_impute_strategy='most_frequent', boolean_fill_value=None, random_seed=0, **kwargs*)

Imputes missing data according to a specified imputation strategy.

**Parameters**

- **categorical_impute_strategy** (*string*) – Impute strategy to use for string, object, boolean, categorical dtypes. Valid values include "most_frequent" and "constant".
- **numeric_impute_strategy** (*string*) – Impute strategy to use for numeric columns. Valid values include "mean", "median", "most_frequent", and "constant".
- **boolean_impute_strategy** (*string*) – Impute strategy to use for boolean columns. Valid values include "most_frequent" and "constant".
- **categorical_fill_value** (*string*) – When categorical_impute_strategy == "constant", fill_value is used to replace missing data. The default value of None will fill with the string "missing_value".
- **numeric_fill_value** (*int, float*) – When numeric_impute_strategy == "constant", fill_value is used to replace missing data. The default value of None will fill with 0.
- **boolean_fill_value** (*bool*) – When boolean_impute_strategy == "constant", fill_value is used to replace missing data. The default value of None will fill with True.
- **random_seed** (*int*) – Seed for the random number generator. Defaults to 0.

**Attributes**

| | |
|---|---|
| **hyper-parameter_ranges** | { "categorical_impute_strategy": ["most_frequent"], "numeric_impute_strategy": ["mean", "median", "most_frequent", "knn"], "boolean_impute_strategy": ["most_frequent"]} |
| **modifies_features** | True |
| **modifies_target** | False |
| **name** | Imputer |
| **training_only** | False |

**Methods**

| | |
|---|---|
| *clone* | Constructs a new component with the same parameters and random state. |
| *default_parameters* | Returns the default parameters for this component. |
| *describe* | Describe a component and its parameters. |
| *fit* | Fits imputer to data. 'None' values are converted to np.nan before imputation and are treated as the same. |
| *fit_transform* | Fits on X and transforms X. |
| *load* | Loads component at file path. |
| *needs_fitting* | Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances. |
| *parameters* | Returns the parameters which were used to initialize the component. |
| *save* | Saves component at file path. |
| *transform* | Transforms data X by imputing missing values. |
| *update_parameters* | Updates the parameter dictionary of the component. |

**clone**(*self*)

Constructs a new component with the same parameters and random state.

**Returns** A new instance of this component with identical parameters and random state.

**default_parameters**(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

**Returns** Default parameters for this component.

**Return type** dict

**describe**(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

**Parameters**

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description as dictionary in the format {"name": name, "parameters": parameters}

**Returns** Returns dictionary if return_dict is True, else None.

**Return type** None or dict

**fit**(*self*, *X*, *y=None*)

Fits imputer to data. 'None' values are converted to np.nan before imputation and are treated as the same.

**Parameters**

- **X** (*pd.DataFrame, np.ndarray*) – The input training data of shape [n_samples, n_features]
- **y** (*pd.Series, optional*) – The target training data of length [n_samples]

**Returns** self

**fit_transform**(*self*, *X*, *y=None*)

Fits on X and transforms X.

**Parameters**

- **X** (*pd.DataFrame*) – Data to fit and transform.
- **y** (*pd.Series*) – Target data.

**Returns** Transformed X.

**Return type** pd.DataFrame

**Raises** **MethodPropertyNotFoundError** – If transformer does not have a transform method or a component_obj that implements transform.

**static load**(*file_path*)

Loads component at file path.

**Parameters** **file_path** (*str*) – Location to load file.

**Returns** ComponentBase object

**needs_fitting**(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need to be fit or whose fit methods do nothing.

**Returns** True.

**property parameters**(*self*)

Returns the parameters which were used to initialize the component.

**save**(*self*, *file_path*, *pickle_protocol=cloudpickle.DEFAULT_PROTOCOL*)

Saves component at file path.

**Parameters**

- **file_path** (*str*) – Location to save file.
- **pickle_protocol** (*int*) – The pickle data stream format.

**transform**(*self*, *X*, *y=None*)

Transforms data X by imputing missing values.

**Parameters**

- **X** (*pd.DataFrame*) – Data to transform
- **y** (*pd.Series, optional*) – Ignored.

**Returns** Transformed X

**Return type** pd.DataFrame

**update_parameters**(*self*, *update_dict*, *reset_fit=True*)

Updates the parameter dictionary of the component.

**Parameters**

- **update_dict** (*dict*) – A dict of parameters to update.
- **reset_fit** (*bool, optional*) – If True, will set *_is_fitted* to False.

**class** evalml.pipelines.components.transformers.imputers.**KNNImputer**(*number_neighbors=3*, *random_seed=0*, ***kwargs*)

Imputes missing data using KNN according to a specified number of neighbors. Natural language columns are ignored.

**Parameters**

- **number_neighbors** (*int*) – Number of nearest neighbors for KNN to search for. Defaults to 3.
- **random_seed** (*int*) – Seed for the random number generator. Defaults to 0.

**Attributes**

| | |
|---|---|
| **modifies_features** | True |
| **modifies_target** | False |
| **name** | KNN Imputer |
| **training_only** | False |

**Methods**

| | |
|---|---|
| *clone* | Constructs a new component with the same parameters and random state. |
| *default_parameters* | Returns the default parameters for this component. |
| *describe* | Describe a component and its parameters. |
| *fit* | Fits imputer to data. 'None' values are converted to np.nan before imputation and are treated as the same. |
| *fit_transform* | Fits on X and transforms X. |
| *load* | Loads component at file path. |
| *needs_fitting* | Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances. |
| *parameters* | Returns the parameters which were used to initialize the component. |
| *save* | Saves component at file path. |
| *transform* | Transforms input by imputing missing values. 'None' and np.nan values are treated as the same. |
| *update_parameters* | Updates the parameter dictionary of the component. |

**clone**(*self*)

Constructs a new component with the same parameters and random state.

**Returns** A new instance of this component with identical parameters and random state.

**default_parameters**(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

**Returns** Default parameters for this component.

**Return type** dict

**describe**(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

**Parameters**

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description as dictionary in the format {"name": name, "parameters": parameters}

**Returns** Returns dictionary if return_dict is True, else None.

**Return type** None or dict

**fit**(*self*, *X*, *y=None*)

Fits imputer to data. 'None' values are converted to np.nan before imputation and are treated as the same.

**Parameters**

- **X** (*pd.DataFrame or np.ndarray*) – the input training data of shape [n_samples, n_features]
- **y** (*pd.Series, optional*) – the target training data of length [n_samples]

**Returns** self

**Raises** **ValueError** – if the KNNImputer receives a dataframe with both Boolean and Categorical data.

**fit_transform**(*self*, *X*, *y=None*)

Fits on X and transforms X.

**Parameters**

- **X** (*pd.DataFrame*) – Data to fit and transform
- **y** (*pd.Series, optional*) – Target data.

**Returns** Transformed X

**Return type** pd.DataFrame

**static load**(*file_path*)

Loads component at file path.

**Parameters** **file_path** (*str*) – Location to load file.

**Returns** ComponentBase object

**needs_fitting**(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need to be fit or whose fit methods do nothing.

**Returns** True.

**property parameters**(*self*)

Returns the parameters which were used to initialize the component.

**save**(*self*, *file_path*, *pickle_protocol=cloudpickle.DEFAULT_PROTOCOL*)

Saves component at file path.

**Parameters**

- **file_path** (*str*) – Location to save file.
- **pickle_protocol** (*int*) – The pickle data stream format.

**transform**(*self*, *X*, *y=None*)

Transforms input by imputing missing values. 'None' and np.nan values are treated as the same.

**Parameters**

- **X** (*pd.DataFrame*) – Data to transform.
- **y** (*pd.Series, optional*) – Ignored.

**Returns** Transformed X

**Return type** pd.DataFrame

**update_parameters**(*self*, *update_dict*, *reset_fit=True*)

Updates the parameter dictionary of the component.

**Parameters**

- **update_dict** (*dict*) – A dict of parameters to update.
- **reset_fit** (*bool, optional*) – If True, will set *_is_fitted* to False.

**class** evalml.pipelines.components.transformers.imputers.**PerColumnImputer**(*impute_strategies=None*, *random_seed=0*, ***kwargs*)

Imputes missing data according to a specified imputation strategy per column.

**Parameters**

- **impute_strategies** (*dict*) – Column and {"impute_strategy": strategy, "fill_value":value} pairings. Valid values for impute strategy include "mean", "median", "most_frequent", "constant" for numerical data, and "most_frequent", "constant" for object data types. Defaults to None, which uses "most_frequent" for all columns. When impute_strategy == "constant", fill_value is used to replace missing data. When None, uses 0 when imputing numerical data and "missing_value" for strings or object data types.
- **random_seed** (*int*) – Seed for the random number generator. Defaults to 0.

**Attributes**

| | |
|---|---|
| **hyper-parameter_ranges** | {} |
| **modifies_features** | True |
| **modifies_target** | False |
| **name** | Per Column Imputer |
| **training_only** | False |

**Methods**

| | |
|---|---|
| *clone* | Constructs a new component with the same parameters and random state. |
| *default_parameters* | Returns the default parameters for this component. |
| *describe* | Describe a component and its parameters. |
| *fit* | Fits imputers on input data. |
| *fit_transform* | Fits on X and transforms X. |
| *load* | Loads component at file path. |
| *needs_fitting* | Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances. |
| *parameters* | Returns the parameters which were used to initialize the component. |
| *save* | Saves component at file path. |
| *transform* | Transforms input data by imputing missing values. |
| *update_parameters* | Updates the parameter dictionary of the component. |

**clone**(*self*)

Constructs a new component with the same parameters and random state.

**Returns** A new instance of this component with identical parameters and random state.

**default_parameters**(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

**Returns** Default parameters for this component.

**Return type** dict

**describe**(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

**Parameters**

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description as dictionary in the format {"name": name, "parameters": parameters}

**Returns** Returns dictionary if return_dict is True, else None.

**Return type** None or dict

**fit**(*self*, *X*, *y=None*)

Fits imputers on input data.

**Parameters**

- **X** (*pd.DataFrame or np.ndarray*) – The input training data of shape [n_samples, n_features] to fit.
- **y** (*pd.Series, optional*) – The target training data of length [n_samples]. Ignored.

**Returns** self

**fit_transform**(*self*, *X*, *y=None*)

Fits on X and transforms X.

**Parameters**

- **X** (*pd.DataFrame*) – Data to fit and transform.
- **y** (*pd.Series*) – Target data.

**Returns** Transformed X.

**Return type** pd.DataFrame

**Raises** **MethodPropertyNotFoundError** – If transformer does not have a transform method or a component_obj that implements transform.

**static load**(*file_path*)

Loads component at file path.

**Parameters** **file_path** (*str*) – Location to load file.

**Returns** ComponentBase object

**needs_fitting**(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need to be fit or whose fit methods do nothing.

**Returns** True.

**property parameters**(*self*)

Returns the parameters which were used to initialize the component.

**save**(*self*, *file_path*, *pickle_protocol=cloudpickle.DEFAULT_PROTOCOL*)

Saves component at file path.

**Parameters**

- **file_path** (*str*) – Location to save file.
- **pickle_protocol** (*int*) – The pickle data stream format.

**transform**(*self*, *X*, *y=None*)

Transforms input data by imputing missing values.

**Parameters**

- **X** (*pd.DataFrame or np.ndarray*) – The input training data of shape [n_samples, n_features] to transform.
- **y** (*pd.Series, optional*) – The target training data of length [n_samples]. Ignored.

**Returns** Transformed X

**Return type** pd.DataFrame

**update_parameters**(*self*, *update_dict*, *reset_fit=True*)

Updates the parameter dictionary of the component.

**Parameters**

- **update_dict** (*dict*) – A dict of parameters to update.
- **reset_fit** (*bool, optional*) – If True, will set *_is_fitted* to False.

**class** evalml.pipelines.components.transformers.imputers.**SimpleImputer**(*impute_strategy='most_frequent', fill_value=None, random_seed=0, **kwargs*)

Imputes missing data according to a specified imputation strategy. Natural language columns are ignored.

**Parameters**

- **impute_strategy** (*string*) – Impute strategy to use. Valid values include "mean", "median", "most_frequent", "constant" for numerical data, and "most_frequent", "constant" for object data types.
- **fill_value** (*string*) – When impute_strategy == "constant", fill_value is used to replace missing data. Defaults to 0 when imputing numerical data and "missing_value" for strings or object data types.
- **random_seed** (*int*) – Seed for the random number generator. Defaults to 0.

**Attributes**

| | |
|---|---|
| **hyper-parameter_ranges** | { "impute_strategy": ["mean", "median", "most_frequent"]} |
| **modifies_features** | True |
| **modifies_target** | False |
| **name** | Simple Imputer |
| **training_only** | False |

**Methods**

| | |
|---|---|
| *clone* | Constructs a new component with the same parameters and random state. |
| *default_parameters* | Returns the default parameters for this component. |
| *describe* | Describe a component and its parameters. |
| *fit* | Fits imputer to data. 'None' values are converted to np.nan before imputation and are treated as the same. |
| *fit_transform* | Fits on X and transforms X. |
| *load* | Loads component at file path. |
| *needs_fitting* | Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances. |
| *parameters* | Returns the parameters which were used to initialize the component. |
| *save* | Saves component at file path. |
| *transform* | Transforms input by imputing missing values. 'None' and np.nan values are treated as the same. |
| *update_parameters* | Updates the parameter dictionary of the component. |

**clone**(*self*)

Constructs a new component with the same parameters and random state.

**Returns** A new instance of this component with identical parameters and random state.

**default_parameters**(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

**Returns** Default parameters for this component.

**Return type** dict

**describe**(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

**Parameters**

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description as dictionary in the format {"name": name, "parameters": parameters}

**Returns** Returns dictionary if return_dict is True, else None.

**Return type** None or dict

**fit**(*self*, *X*, *y=None*)

Fits imputer to data. 'None' values are converted to np.nan before imputation and are treated as the same.

**Parameters**

- **X** (*pd.DataFrame or np.ndarray*) – the input training data of shape [n_samples, n_features]
- **y** (*pd.Series, optional*) – the target training data of length [n_samples]

**Returns** self

**Raises** **ValueError** – if the SimpleImputer receives a dataframe with both Boolean and Categorical data.

**fit_transform**(*self*, *X*, *y=None*)

Fits on X and transforms X.

**Parameters**

- **X** (*pd.DataFrame*) – Data to fit and transform
- **y** (*pd.Series, optional*) – Target data.

**Returns** Transformed X

**Return type** pd.DataFrame

**static load**(*file_path*)

Loads component at file path.

**Parameters** **file_path** (*str*) – Location to load file.

**Returns** ComponentBase object

**needs_fitting**(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need to be fit or whose fit methods do nothing.

**Returns** True.

**property parameters**(*self*)

Returns the parameters which were used to initialize the component.

**save**(*self*, *file_path*, *pickle_protocol=cloudpickle.DEFAULT_PROTOCOL*)

Saves component at file path.

**Parameters**

- **file_path** (*str*) – Location to save file.
- **pickle_protocol** (*int*) – The pickle data stream format.

**transform**(*self*, *X*, *y=None*)

Transforms input by imputing missing values. 'None' and np.nan values are treated as the same.

**Parameters**

- **X** (*pd.DataFrame*) – Data to transform.
- **y** (*pd.Series, optional*) – Ignored.

**Returns** Transformed X

**Return type** pd.DataFrame

**update_parameters**(*self*, *update_dict*, *reset_fit=True*)

Updates the parameter dictionary of the component.

**Parameters**

- **update_dict** (*dict*) – A dict of parameters to update.
- **reset_fit** (*bool, optional*) – If True, will set *_is_fitted* to False.

**class** evalml.pipelines.components.transformers.imputers.**TargetImputer**(*impute_strategy='most_frequent'*, *fill_value=None*, *random_seed=0*, ***kwargs*)

Imputes missing target data according to a specified imputation strategy.

**Parameters**

- **impute_strategy** (*string*) – Impute strategy to use. Valid values include "mean", "median", "most_frequent", "constant" for numerical data, and "most_frequent", "constant" for object data types. Defaults to "most_frequent".
- **fill_value** (*string*) – When impute_strategy == "constant", fill_value is used to replace missing data. Defaults to None which uses 0 when imputing numerical data and "missing_value" for strings or object data types.
- **random_seed** (*int*) – Seed for the random number generator. Defaults to 0.

**Attributes**

| | |
|---|---|
| **hyper-parameter_ranges** | { "impute_strategy": ["mean", "median", "most_frequent"]} |
| **modifies_features** | False |
| **modifies_target** | True |
| **name** | Target Imputer |
| **training_only** | False |

**Methods**

| | |
|---|---|
| *clone* | Constructs a new component with the same parameters and random state. |
| *default_parameters* | Returns the default parameters for this component. |
| *describe* | Describe a component and its parameters. |
| *fit* | Fits imputer to target data. 'None' values are converted to np.nan before imputation and are treated as the same. |
| *fit_transform* | Fits on and transforms the input target data. |
| *load* | Loads component at file path. |
| *needs_fitting* | Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances. |
| *parameters* | Returns the parameters which were used to initialize the component. |
| *save* | Saves component at file path. |
| *transform* | Transforms input target data by imputing missing values. 'None' and np.nan values are treated as the same. |
| *update_parameters* | Updates the parameter dictionary of the component. |

**clone**(*self*)

Constructs a new component with the same parameters and random state.

**Returns** A new instance of this component with identical parameters and random state.

**default_parameters**(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

**Returns** Default parameters for this component.

**Return type** dict

**describe**(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

**Parameters**

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description as dictionary in the format {"name": name, "parameters": parameters}

**Returns** Returns dictionary if return_dict is True, else None.

**Return type** None or dict

`fit`(*self*, *X*, *y*)

Fits imputer to target data. 'None' values are converted to np.nan before imputation and are treated as the same.

**Parameters**

- **X** (*pd.DataFrame or np.ndarray*) – The input training data of shape [n_samples, n_features]. Ignored.
- **y** (*pd.Series, optional*) – The target training data of length [n_samples].

**Returns** self

**Raises** **TypeError** – If target is filled with all null values.

`fit_transform`(*self*, *X*, *y*)

Fits on and transforms the input target data.

**Parameters**

- **X** (*pd.DataFrame*) – Features. Ignored.
- **y** (*pd.Series*) – Target data to impute.

**Returns** The original X, transformed y

**Return type** (pd.DataFrame, pd.Series)

`static load`(*file_path*)

Loads component at file path.

**Parameters** **file_path** (*str*) – Location to load file.

**Returns** ComponentBase object

`needs_fitting`(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need to be fit or whose fit methods do nothing.

**Returns** True.

`property parameters`(*self*)

Returns the parameters which were used to initialize the component.

`save`(*self*, *file_path*, *pickle_protocol=cloudpickle.DEFAULT_PROTOCOL*)

Saves component at file path.

**Parameters**

- **file_path** (*str*) – Location to save file.
- **pickle_protocol** (*int*) – The pickle data stream format.

`transform`(*self*, *X*, *y*)

Transforms input target data by imputing missing values. 'None' and np.nan values are treated as the same.

**Parameters**

- **X** (*pd.DataFrame*) – Features. Ignored.

- **y** (*pd.Series*) – Target data to impute.

**Returns** The original X, transformed y

**Return type** (pd.DataFrame, pd.Series)

**update_parameters**(*self*, *update_dict*, *reset_fit=True*)

Updates the parameter dictionary of the component.

**Parameters**

- **update_dict** (*dict*) – A dict of parameters to update.
- **reset_fit** (*bool, optional*) – If True, will set *_is_fitted* to False.

**class** evalml.pipelines.components.transformers.imputers.**TimeSeriesImputer**(*categorical_impute_strategy='forwara nu-meric_impute_strategy='interpolate', tar-get_impute_strategy='forwards_fill', random_seed=0, **kwargs*)

Imputes missing data according to a specified timeseries-specific imputation strategy.

This Transformer should be used after the *TimeSeriesRegularizer* in order to impute the missing values that were added to X and y (if passed).

**Parameters**

- **categorical_impute_strategy** (*string*) – Impute strategy to use for string, object, boolean, categorical dtypes. Valid values include "backwards_fill" and "forwards_fill". Defaults to "forwards_fill".
- **numeric_impute_strategy** (*string*) – Impute strategy to use for numeric columns. Valid values include "backwards_fill", "forwards_fill", and "interpolate". Defaults to "interpolate".
- **target_impute_strategy** (*string*) – Impute strategy to use for the target column. Valid values include "backwards_fill", "forwards_fill", and "interpolate". Defaults to "forwards_fill".
- **random_seed** (*int*) – Seed for the random number generator. Defaults to 0.

**Raises** **ValueError** – If categorical_impute_strategy, numeric_impute_strategy, or target_impute_strategy is not one of the valid values.

**Attributes**

| | |
|---|---|
| **hyper-parameter_ranges** | { "categorical_impute_strategy": ["backwards_fill", "forwards_fill"], "numeric_impute_strategy": ["backwards_fill", "forwards_fill", "interpolate"], "target_impute_strategy": ["backwards_fill", "forwards_fill", "interpolate"],} |
| **modifies_features** | True |
| **modifies_target** | True |
| **name** | Time Series Imputer |
| **training_only** | True |

**Methods**

| | |
|---|---|
| *clone* | Constructs a new component with the same parameters and random state. |
| *default_parameters* | Returns the default parameters for this component. |
| *describe* | Describe a component and its parameters. |
| *fit* | Fits imputer to data. |
| *fit_transform* | Fits on X and transforms X. |
| *load* | Loads component at file path. |
| *needs_fitting* | Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances. |
| *parameters* | Returns the parameters which were used to initialize the component. |
| *save* | Saves component at file path. |
| *transform* | Transforms data X by imputing missing values using specified timeseries-specific strategies. 'None' values are converted to np.nan before imputation and are treated as the same. |
| *update_parameters* | Updates the parameter dictionary of the component. |

**clone**(*self*)

Constructs a new component with the same parameters and random state.

**Returns** A new instance of this component with identical parameters and random state.

**default_parameters**(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

**Returns** Default parameters for this component.

**Return type** dict

**describe**(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

**Parameters**

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description as dictionary in the format {"name": name, "parameters": parameters}

**Returns** Returns dictionary if return_dict is True, else None.

**Return type** None or dict

**fit**(*self*, *X*, *y=None*)

Fits imputer to data.

'None' values are converted to np.nan before imputation and are treated as the same. If a value is missing at the beginning or end of a column, that value will be imputed using backwards fill or forwards fill as necessary, respectively.

**Parameters**

- **X** (*pd.DataFrame, np.ndarray*) – The input training data of shape [n_samples, n_features]
- **y** (*pd.Series, optional*) – The target training data of length [n_samples]

**Returns** self

**fit_transform**(*self*, *X*, *y=None*)

Fits on X and transforms X.

**Parameters**

- **X** (*pd.DataFrame*) – Data to fit and transform.
- **y** (*pd.Series*) – Target data.

**Returns** Transformed X.

**Return type** pd.DataFrame

**Raises** **MethodPropertyNotFoundError** – If transformer does not have a transform method or a component_obj that implements transform.

**static load**(*file_path*)

Loads component at file path.

**Parameters** **file_path** (*str*) – Location to load file.

**Returns** ComponentBase object

**needs_fitting**(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need to be fit or whose fit methods do nothing.

**Returns** True.

**property parameters**(*self*)

Returns the parameters which were used to initialize the component.

**save**(*self*, *file_path*, *pickle_protocol=cloudpickle.DEFAULT_PROTOCOL*)

Saves component at file path.

**Parameters**

- **file_path** (*str*) – Location to save file.
- **pickle_protocol** (*int*) – The pickle data stream format.

**transform**(*self*, *X*, *y=None*)

Transforms data X by imputing missing values using specified timeseries-specific strategies. 'None' values are converted to np.nan before imputation and are treated as the same.

**Parameters**

- **X** (*pd.DataFrame*) – Data to transform.
- **y** (*pd.Series, optional*) – Optionally, target data to transform.

**Returns** Transformed X and y

**Return type** pd.DataFrame

**update_parameters**(*self*, *update_dict*, *reset_fit=True*)

Updates the parameter dictionary of the component.

**Parameters**

- **update_dict** (*dict*) – A dict of parameters to update.
- **reset_fit** (*bool, optional*) – If True, will set *_is_fitted* to False.

### preprocessing

Preprocessing transformer components.

### Submodules

### datetime_featurizer

Transformer that can automatically extract features from datetime columns.

### Module Contents

### Classes Summary

| | |
|---|---|
| *DateTimeFeaturizer* | Transformer that can automatically extract features from datetime columns. |

### Contents

**class** evalml.pipelines.components.transformers.preprocessing.datetime_featurizer.**DateTimeFeaturizer**(*featu en- code time ran- dom **kw*

Transformer that can automatically extract features from datetime columns.

**Parameters**

- **features_to_extract** (*list*) – List of features to extract. Valid options include "year", "month", "day_of_week", "hour". Defaults to None.
- **encode_as_categories** (*bool*) – Whether day-of-week and month features should be encoded as pandas "category" dtype. This allows OneHotEncoders to encode these features. Defaults to False.
- **time_index** (*str*) – Name of the column containing the datetime information used to order the data. Ignored.
- **random_seed** (*int*) – Seed for the random number generator. Defaults to 0.

**Attributes**

| | |
|---|---|
| **hyper-parameter_ranges** | {} |
| **modifies_features** | True |
| **modifies_target** | False |
| **name** | DateTime Featurizer |
| **training_only** | False |

**Methods**

| | |
|---|---|
| *clone* | Constructs a new component with the same parameters and random state. |
| *default_parameters* | Returns the default parameters for this component. |
| *describe* | Describe a component and its parameters. |
| *fit* | Fit the datetime featurizer component. |
| *fit_transform* | Fits on X and transforms X. |
| *get_feature_names* | Gets the categories of each datetime feature. |
| *load* | Loads component at file path. |
| *needs_fitting* | Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances. |
| *parameters* | Returns the parameters which were used to initialize the component. |
| *save* | Saves component at file path. |
| *transform* | Transforms data X by creating new features using existing DateTime columns, and then dropping those DateTime columns. |
| *update_parameters* | Updates the parameter dictionary of the component. |

**clone**(*self*)

Constructs a new component with the same parameters and random state.

**Returns** A new instance of this component with identical parameters and random state.

**default_parameters**(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

**Returns** Default parameters for this component.

**Return type** dict

**describe**(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

**Parameters**

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description as dictionary in the format {"name": name, "parameters": parameters}

**Returns** Returns dictionary if return_dict is True, else None.

**Return type** None or dict

`fit`(*self*, *X*, *y=None*)

Fit the datetime featurizer component.

**Parameters**

- **X** (*pd.DataFrame*) – Input features.
- **y** (*pd.Series, optional*) – Target data. Ignored.

**Returns** self

`fit_transform`(*self*, *X*, *y=None*)

Fits on X and transforms X.

**Parameters**

- **X** (*pd.DataFrame*) – Data to fit and transform.
- **y** (*pd.Series*) – Target data.

**Returns** Transformed X.

**Return type** pd.DataFrame

**Raises** **MethodPropertyNotFoundError** – If transformer does not have a transform method or a component_obj that implements transform.

`get_feature_names`(*self*)

Gets the categories of each datetime feature.

**Returns**

**Dictionary, where each key-value pair is a column name and a dictionary** mapping the unique feature values to their integer encoding.

**Return type** dict

`static load`(*file_path*)

Loads component at file path.

**Parameters** **file_path** (*str*) – Location to load file.

**Returns** ComponentBase object

`needs_fitting`(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need to be fit or whose fit methods do nothing.

**Returns** True.

`property parameters`(*self*)

Returns the parameters which were used to initialize the component.

`save`(*self*, *file_path*, *pickle_protocol=cloudpickle.DEFAULT_PROTOCOL*)

Saves component at file path.

**Parameters**

- **file_path** (*str*) – Location to save file.

- **pickle_protocol** (*int*) – The pickle data stream format.

**transform**(*self*, *X*, *y=None*)

Transforms data X by creating new features using existing DateTime columns, and then dropping those DateTime columns.

**Parameters**

- **X** (*pd.DataFrame*) – Input features.
- **y** (*pd.Series, optional*) – Ignored.

**Returns** Transformed X

**Return type** pd.DataFrame

**update_parameters**(*self*, *update_dict*, *reset_fit=True*)

Updates the parameter dictionary of the component.

**Parameters**

- **update_dict** (*dict*) – A dict of parameters to update.
- **reset_fit** (*bool, optional*) – If True, will set *_is_fitted* to False.

## decomposer

Component that removes trends from time series and returns the decomposed components.

### Module Contents

### Classes Summary

| | |
|---|---|
| *Decomposer* | Component that removes trends and seasonality from time series and returns the decomposed components. |

### Contents

**class** evalml.pipelines.components.transformers.preprocessing.decomposer.**Decomposer**(*component_obj=None*, *random_seed: int = 0*, *degree: int = 1*, *period: int = -1*, *seasonal_smoother: int = 7*, *time_index: str = None*, ***kwargs*)

Component that removes trends and seasonality from time series and returns the decomposed components.

**Parameters**

- **parameters** (*dict*) – Dictionary of parameters to pass to component object.
- **component_obj** (*class*) – Instance of a detrender/deseasonalizer class.
- **random_seed** (*int*) – Seed for the random number generator. Defaults to 0.
- **degree** (*int*) – Currently the degree of the PolynomialDecomposer, not used for STLDecomposer.
- **period** (*int*) – The best guess, in units, for the period of the seasonal signal.
- **seasonal_smoother** (*int*) – The seasonal smoothing parameter for STLDecomposer, not used for PolynomialDecomposer.
- **time_index** (*str*) – The column name of the feature matrix (X) that the datetime information should be pulled from.

**Attributes**

| | |
|---|---|
| **hyperparameter_ranges** | None |
| **invalid_frequencies** | [] |
| **modifies_features** | False |
| **modifies_target** | True |
| **name** | Decomposer |
| **needs_fitting** | True |
| **training_only** | False |

**Methods**

| | |
|---|---|
| clone | Constructs a new component with the same parameters and random state. |
| default_parameters | Returns the default parameters for this component. |
| describe | Describe a component and its parameters. |
| determine_periodicity | Function that uses autocorrelative methods to determine the likely most signficant period of the seasonal signal. |
| fit | Fits component to data. |
| fit_transform | Removes fitted trend and seasonality from target variable. |
| get_trend_dataframe | Return a list of dataframes, each with 3 columns: trend, seasonality, residual. |
| inverse_transform | Add the trend + seasonality back to y. |
| is_freq_valid | Determines if the given string represents a valid frequency for this decomposer. |
| load | Loads component at file path. |
| parameters | Returns the parameters which were used to initialize the component. |
| plot_decomposition | Plots the decomposition of the target signal. |
| save | Saves component at file path. |
| set_period | Function to set the component's seasonal period based on the target's seasonality. |
| transform | Transforms data X. |
| update_parameters | Updates the parameter dictionary of the component. |

**clone**(*self*)

Constructs a new component with the same parameters and random state.

**Returns** A new instance of this component with identical parameters and random state.

**default_parameters**(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

**Returns** Default parameters for this component.

**Return type** dict

**describe**(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

**Parameters**

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description as dictionary in the format {"name": name, "parameters": parameters}

**Returns** Returns dictionary if return_dict is True, else None.

**Return type** None or dict

**classmethod determine_periodicity**(*cls*, *X: pandas.DataFrame*, *y: pandas.Series*, *acf_threshold: float = 0.01*, *rel_max_order: int = 5*)

Function that uses autocorrelative methods to determine the likely most signficant period of the seasonal signal.

**Parameters**

- **X** (*pandas.DataFrame*) – The feature data of the time series problem.
- **y** (*pandas.Series*) – The target data of a time series problem.
- **acf_threshold** (*float*) – The threshold for the autocorrelation function to determine the period. Any values below the threshold are considered to be 0 and will not be considered for the period. Defaults to 0.01.
- **rel_max_order** (*int*) – The order of the relative maximum to determine the period. Defaults to 5.

**Returns**

**The integer number of entries in time series data over which the seasonal part of the target data** repeats. If the time series data is in days, then this is the number of days that it takes the target's seasonal signal to repeat. Note: the target data can contain multiple seasonal signals. This function will only return the stronger. E.g. if the target has both weekly and yearly seasonality, the function may return either "7" or "365", depending on which seasonality is more strongly autocorrelated. If no period is detected, returns None.

**Return type** int

**fit**(*self*, *X*, *y=None*)

Fits component to data.

**Parameters**

- **X** (*pd.DataFrame*) – The input training data of shape [n_samples, n_features]
- **y** (*pd.Series, optional*) – The target training data of length [n_samples]

**Returns** self

**Raises** **MethodPropertyNotFoundError** – If component does not have a fit method or a component_obj that implements fit.

**fit_transform**(*self*, *X: pandas.DataFrame*, *y: pandas.Series = None*) → tuple[pandas.DataFrame, pandas.Series]

Removes fitted trend and seasonality from target variable.

**Parameters**

- **X** (*pd.DataFrame, optional*) – Ignored.
- **y** (*pd.Series*) – Target variable to detrend and deseasonalize.

**Returns**

**The first element are the input features returned without modification.** The second element is the target variable y with the fitted trend removed.

**Return type** tuple of pd.DataFrame, pd.Series

**abstract get_trend_dataframe**(*self*, *y: pandas.Series*)

Return a list of dataframes, each with 3 columns: trend, seasonality, residual.

**abstract inverse_transform**(*self*, *y: pandas.Series*)

Add the trend + seasonality back to y.

**classmethod is_freq_valid**(*cls*, *freq: str*)

Determines if the given string represents a valid frequency for this decomposer.

**Parameters** **freq** (*str*) – A frequency to validate. See the pandas docs at https://pandas.pydata.org/pandas-docs/stable/user_guide/timeseries.html#offset-aliases for options.

**Returns** boolean representing whether the frequency is valid or not.

**static load**(*file_path*)

Loads component at file path.

**Parameters** **file_path** (*str*) – Location to load file.

**Returns** ComponentBase object

**property parameters**(*self*)

Returns the parameters which were used to initialize the component.

**plot_decomposition**(*self*, *X: pandas.DataFrame*, *y: pandas.Series*, *show: bool = False*) → tuple[matplotlib.pyplot.Figure, list]

Plots the decomposition of the target signal.

**Parameters**

- **X** (*pd.DataFrame*) – Input data with time series data in index.
- **y** (*pd.Series or pd.DataFrame*) – Target variable data provided as a Series for univariate problems or a DataFrame for multivariate problems.
- **show** (*bool*) – Whether to display the plot or not. Defaults to False.

**Returns**

**The figure and axes that have the decompositions** plotted on them

**Return type** matplotlib.pyplot.Figure, list[matplotlib.pyplot.Axes]

**save**(*self*, *file_path*, *pickle_protocol=cloudpickle.DEFAULT_PROTOCOL*)

Saves component at file path.

**Parameters**

- **file_path** (*str*) – Location to save file.
- **pickle_protocol** (*int*) – The pickle data stream format.

**set_period**(*self*, *X: pandas.DataFrame*, *y: pandas.Series*, *acf_threshold: float = 0.01*, *rel_max_order: int = 5*)

Function to set the component's seasonal period based on the target's seasonality.

**Parameters**

- **X** (*pandas.DataFrame*) – The feature data of the time series problem.
- **y** (*pandas.Series*) – The target data of a time series problem.
- **acf_threshold** (*float*) – The threshold for the autocorrelation function to determine the period. Any values below the threshold are considered to be 0 and will not be considered for the period. Defaults to 0.01.
- **rel_max_order** (*int*) – The order of the relative maximum to determine the period. Defaults to 5.

**abstract transform**(*self*, *X*, *y=None*)

Transforms data X.

**Parameters**

- **X** (*pd.DataFrame*) – Data to transform.
- **y** (*pd.Series, optional*) – Target data.

**Returns** Transformed X

**Return type** pd.DataFrame

**Raises** **MethodPropertyNotFoundError** – If transformer does not have a transform method or a component_obj that implements transform.

**update_parameters**(*self*, *update_dict*, *reset_fit=True*)

Updates the parameter dictionary of the component.

**Parameters**

- **update_dict** (*dict*) – A dict of parameters to update.
- **reset_fit** (*bool, optional*) – If True, will set *_is_fitted* to False.

## drop_nan_rows_transformer

Transformer to drop rows specified by row indices.

### Module Contents

### Classes Summary

| | |
|---|---|
| *DropNaNRowsTransformer* | Transformer to drop rows with NaN values. |

### Contents

**class** evalml.pipelines.components.transformers.preprocessing.drop_nan_rows_transformer.**DropNaNRowsTrans**

Transformer to drop rows with NaN values.

**Parameters** **random_seed** (*int*) – Seed for the random number generator. Is not used by this component. Defaults to 0.

**Attributes**

| | |
|---|---|
| **hyper-parameter_ranges** | {} |
| **modifies_features** | True |
| **modifies_target** | True |
| **name** | Drop NaN Rows Transformer |
| **training_only** | False |

**Methods**

| | |
|---|---|
| *clone* | Constructs a new component with the same parameters and random state. |
| *default_parameters* | Returns the default parameters for this component. |
| *describe* | Describe a component and its parameters. |
| *fit* | Fits component to data. |
| *fit_transform* | Fits on X and transforms X. |
| *load* | Loads component at file path. |
| *needs_fitting* | Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances. |
| *parameters* | Returns the parameters which were used to initialize the component. |
| *save* | Saves component at file path. |
| *transform* | Transforms data using fitted component. |
| *update_parameters* | Updates the parameter dictionary of the component. |

**clone**(*self*)

Constructs a new component with the same parameters and random state.

**Returns** A new instance of this component with identical parameters and random state.

**default_parameters**(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

**Returns** Default parameters for this component.

**Return type** dict

**describe**(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

**Parameters**

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description as dictionary in the format {"name": name, "parameters": parameters}

**Returns** Returns dictionary if return_dict is True, else None.

**Return type** None or dict

**fit**(*self*, *X*, *y=None*)

Fits component to data.

**Parameters**

- **X** (*pd.DataFrame*) – The input training data of shape [n_samples, n_features].
- **y** (*pd.Series, optional*) – The target training data of length [n_samples].

**Returns** self

**fit_transform**(*self*, *X*, *y=None*)

Fits on X and transforms X.

**Parameters**

- **X** (*pd.DataFrame*) – Data to fit and transform.
- **y** (*pd.Series*) – Target data.

**Returns** Transformed X.

**Return type** pd.DataFrame

**Raises** **MethodPropertyNotFoundError** – If transformer does not have a transform method or a component_obj that implements transform.

**static load**(*file_path*)

Loads component at file path.

**Parameters** **file_path** (*str*) – Location to load file.

**Returns** ComponentBase object

**needs_fitting**(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need to be fit or whose fit methods do nothing.

**Returns** True.

**property parameters**(*self*)

Returns the parameters which were used to initialize the component.

**save**(*self*, *file_path*, *pickle_protocol=cloudpickle.DEFAULT_PROTOCOL*)

Saves component at file path.

**Parameters**

- **file_path** (*str*) – Location to save file.
- **pickle_protocol** (*int*) – The pickle data stream format.

**transform**(*self*, *X*, *y=None*)

Transforms data using fitted component.

**Parameters**

- **X** (*pd.DataFrame*) – Features.
- **y** (*pd.Series, optional*) – Target data.

**Returns** Data with NaN rows dropped.

**Return type** (pd.DataFrame, pd.Series)

**update_parameters**(*self*, *update_dict*, *reset_fit=True*)

Updates the parameter dictionary of the component.

**Parameters**

- **update_dict** (*dict*) – A dict of parameters to update.
- **reset_fit** (*bool, optional*) – If True, will set *_is_fitted* to False.

### drop_null_columns

Transformer to drop features whose percentage of NaN values exceeds a specified threshold.

### Module Contents

### Classes Summary

| | |
|---|---|
| *DropNullColumns* | Transformer to drop features whose percentage of NaN values exceeds a specified threshold. |

### Contents

**class** evalml.pipelines.components.transformers.preprocessing.drop_null_columns.**DropNullColumns**(*pct_null_th ran- dom_seed= **kwargs*)

Transformer to drop features whose percentage of NaN values exceeds a specified threshold.

**Parameters**

- **pct_null_threshold** (*float*) – The percentage of NaN values in an input feature to drop. Must be a value between [0, 1] inclusive. If equal to 0.0, will drop columns with any null values. If equal to 1.0, will drop columns with all null values. Defaults to 0.95.
- **random_seed** (*int*) – Seed for the random number generator. Defaults to 0.

**Attributes**

| | |
|---|---|
| **hyper- parame- ter_ranges** | {} |
| **modi- fies_features** | True |
| **modi- fies_target** | False |
| **name** | Drop Null Columns Transformer |
| **train- ing_only** | False |

**Methods**

| | |
|---|---|
| *clone* | Constructs a new component with the same parameters and random state. |
| *default_parameters* | Returns the default parameters for this component. |
| *describe* | Describe a component and its parameters. |
| *fit* | Fits component to data. |
| *fit_transform* | Fits on X and transforms X. |
| *load* | Loads component at file path. |
| *needs_fitting* | Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances. |
| *parameters* | Returns the parameters which were used to initialize the component. |
| *save* | Saves component at file path. |
| *transform* | Transforms data X by dropping columns that exceed the threshold of null values. |
| *update_parameters* | Updates the parameter dictionary of the component. |

**clone**(*self*)

Constructs a new component with the same parameters and random state.

**Returns** A new instance of this component with identical parameters and random state.

**default_parameters**(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

**Returns** Default parameters for this component.

**Return type** dict

**describe**(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

**Parameters**

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description as dictionary in the format {"name": name, "parameters": parameters}

**Returns** Returns dictionary if return_dict is True, else None.

**Return type** None or dict

**fit**(*self*, *X*, *y=None*)

Fits component to data.

**Parameters**

- **X** (*pd.DataFrame*) – The input training data of shape [n_samples, n_features].
- **y** (*pd.Series, optional*) – The target training data of length [n_samples].

**Returns** self

**fit_transform**(*self*, *X*, *y=None*)

Fits on X and transforms X.

**Parameters**

- **X** (*pd.DataFrame*) – Data to fit and transform.
- **y** (*pd.Series*) – Target data.

**Returns** Transformed X.

**Return type** pd.DataFrame

**Raises** **MethodPropertyNotFoundError** – If transformer does not have a transform method or a component_obj that implements transform.

**static load**(*file_path*)

Loads component at file path.

**Parameters** **file_path** (*str*) – Location to load file.

**Returns** ComponentBase object

**needs_fitting**(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need to be fit or whose fit methods do nothing.

**Returns** True.

**property parameters**(*self*)

Returns the parameters which were used to initialize the component.

**save**(*self*, *file_path*, *pickle_protocol=cloudpickle.DEFAULT_PROTOCOL*)

Saves component at file path.

**Parameters**

- **file_path** (*str*) – Location to save file.
- **pickle_protocol** (*int*) – The pickle data stream format.

**transform**(*self*, *X*, *y=None*)

Transforms data X by dropping columns that exceed the threshold of null values.

**Parameters**

- **X** (*pd.DataFrame*) – Data to transform
- **y** (*pd.Series, optional*) – Ignored.

**Returns** Transformed X

**Return type** pd.DataFrame

**update_parameters**(*self*, *update_dict*, *reset_fit=True*)

Updates the parameter dictionary of the component.

**Parameters**

- **update_dict** (*dict*) – A dict of parameters to update.
- **reset_fit** (*bool, optional*) – If True, will set *_is_fitted* to False.

### drop_rows_transformer

Transformer to drop rows specified by row indices.

### Module Contents

### Classes Summary

| | |
|---|---|
| *DropRowsTransformer* | Transformer to drop rows specified by row indices. |

### Contents

**class** evalml.pipelines.components.transformers.preprocessing.drop_rows_transformer.**DropRowsTransformer**(

Transformer to drop rows specified by row indices.

**Parameters**

- **indices_to_drop** (*list*) – List of indices to drop in the input data. Defaults to None.
- **random_seed** (*int*) – Seed for the random number generator. Is not used by this component. Defaults to 0.

**Attributes**

| | |
|---|---|
| **hyperparameter_ranges** | {} |
| **modifies_features** | True |
| **modifies_target** | True |
| **name** | Drop Rows Transformer |
| **training_only** | True |

**Methods**

| | |
|---|---|
| *clone* | Constructs a new component with the same parameters and random state. |
| *default_parameters* | Returns the default parameters for this component. |
| *describe* | Describe a component and its parameters. |
| *fit* | Fits component to data. |
| *fit_transform* | Fits on X and transforms X. |
| *load* | Loads component at file path. |
| *needs_fitting* | Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances. |
| *parameters* | Returns the parameters which were used to initialize the component. |
| *save* | Saves component at file path. |
| *transform* | Transforms data using fitted component. |
| *update_parameters* | Updates the parameter dictionary of the component. |

**clone**(*self*)

Constructs a new component with the same parameters and random state.

**Returns** A new instance of this component with identical parameters and random state.

**default_parameters**(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

**Returns** Default parameters for this component.

**Return type** dict

**describe**(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

**Parameters**

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description as dictionary in the format {"name": name, "parameters": parameters}

**Returns** Returns dictionary if return_dict is True, else None.

**Return type** None or dict

**fit**(*self*, *X*, *y=None*)

Fits component to data.

**Parameters**

- **X** (*pd.DataFrame*) – The input training data of shape [n_samples, n_features].
- **y** (*pd.Series, optional*) – The target training data of length [n_samples].

**Returns** self

**Raises** **ValueError** – If indices to drop do not exist in input features or target.

**fit_transform**(*self*, *X*, *y=None*)

Fits on X and transforms X.

**Parameters**

- **X** (*pd.DataFrame*) – Data to fit and transform.
- **y** (*pd.Series*) – Target data.

**Returns** Transformed X.

**Return type** pd.DataFrame

**Raises** **MethodPropertyNotFoundError** – If transformer does not have a transform method or a component_obj that implements transform.

**static load**(*file_path*)

Loads component at file path.

**Parameters** **file_path** (*str*) – Location to load file.

**Returns** ComponentBase object

**needs_fitting**(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need to be fit or whose fit methods do nothing.

**Returns** True.

**property parameters**(*self*)

Returns the parameters which were used to initialize the component.

**save**(*self*, *file_path*, *pickle_protocol=cloudpickle.DEFAULT_PROTOCOL*)

Saves component at file path.

**Parameters**

- **file_path** (*str*) – Location to save file.
- **pickle_protocol** (*int*) – The pickle data stream format.

**transform**(*self*, *X*, *y=None*)

Transforms data using fitted component.

**Parameters**

- **X** (*pd.DataFrame*) – Features.
- **y** (*pd.Series, optional*) – Target data.

**Returns** Data with row indices dropped.

**Return type** (pd.DataFrame, pd.Series)

**update_parameters**(*self*, *update_dict*, *reset_fit=True*)

Updates the parameter dictionary of the component.

**Parameters**

- **update_dict** (*dict*) – A dict of parameters to update.
- **reset_fit** (*bool, optional*) – If True, will set *_is_fitted* to False.

### featuretools

Featuretools DFS component that generates features for the input features.

### Module Contents

### Classes Summary

| | |
|---|---|
| *DFSTransformer* | Featuretools DFS component that generates features for the input features. |

### Contents

**class** evalml.pipelines.components.transformers.preprocessing.featuretools.**DFSTransformer**(*index='index', features=None, random_seed=0, **kwargs*)

Featuretools DFS component that generates features for the input features.

**Parameters**

- **index** (*string*) – The name of the column that contains the indices. If no column with this name exists, then featuretools.EntitySet() creates a column with this name to serve as the index column. Defaults to 'index'.
- **random_seed** (*int*) – Seed for the random number generator. Defaults to 0.
- **features** (*list*) – List of features to run DFS on. Defaults to None. Features will only be computed if the columns used by the feature exist in the input and if the feature itself is not in input. If features is an empty list, no transformation will occur to inputted data.

**Attributes**

| | |
|---|---|
| **hyper-parameter_ranges** | {} |
| **modifies_features** | True |
| **modifies_target** | False |
| **name** | DFS Transformer |
| **training_only** | False |

**Methods**

| | |
|---|---|
| *clone* | Constructs a new component with the same parameters and random state. |
| *contains_pre_existing_features* | Determines whether or not features from a DFS Transformer match pipeline input features. |
| *default_parameters* | Returns the default parameters for this component. |
| *describe* | Describe a component and its parameters. |
| *fit* | Fits the DFSTransformer Transformer component. |
| *fit_transform* | Fits on X and transforms X. |
| *load* | Loads component at file path. |
| *needs_fitting* | Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances. |
| *parameters* | Returns the parameters which were used to initialize the component. |
| *save* | Saves component at file path. |
| *transform* | Computes the feature matrix for the input X using featuretools' dfs algorithm. |
| *update_parameters* | Updates the parameter dictionary of the component. |

**clone**(*self*)

Constructs a new component with the same parameters and random state.

**Returns** A new instance of this component with identical parameters and random state.

**static contains_pre_existing_features**(*dfs_features: Optional[List[featuretools.feature_base.FeatureBase]], input_feature_names: List[str], target: Optional[str] = None*)

Determines whether or not features from a DFS Transformer match pipeline input features.

**Parameters**

- **dfs_features** (*Optional[List[FeatureBase]]*) – List of features output from a DFS Transformer.
- **input_feature_names** (*List[str]*) – List of input features into the DFS Transformer.
- **target** (*Optional[str]*) – The target whose values we are trying to predict. This is used to know which column to ignore if the target column is present in the list of features in the DFS Transformer's parameters.

**default_parameters**(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

**Returns** Default parameters for this component.

**Return type** dict

**describe**(*self, print_name=False, return_dict=False*)

Describe a component and its parameters.

**Parameters**

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description as dictionary in the format {"name": name, "parameters": parameters}

**Returns** Returns dictionary if return_dict is True, else None.

**Return type** None or dict

`fit`(*self*, *X*, *y=None*)

Fits the DFSTransformer Transformer component.

**Parameters**

- **X** (*pd.DataFrame, np.array*) – The input data to transform, of shape [n_samples, n_features].
- **y** (*pd.Series*) – The target training data of length [n_samples].

**Returns** self

`fit_transform`(*self*, *X*, *y=None*)

Fits on X and transforms X.

**Parameters**

- **X** (*pd.DataFrame*) – Data to fit and transform.
- **y** (*pd.Series*) – Target data.

**Returns** Transformed X.

**Return type** pd.DataFrame

**Raises** **MethodPropertyNotFoundError** – If transformer does not have a transform method or a component_obj that implements transform.

`static load`(*file_path*)

Loads component at file path.

**Parameters** **file_path** (*str*) – Location to load file.

**Returns** ComponentBase object

`needs_fitting`(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need to be fit or whose fit methods do nothing.

**Returns** True.

`property parameters`(*self*)

Returns the parameters which were used to initialize the component.

`save`(*self*, *file_path*, *pickle_protocol=cloudpickle.DEFAULT_PROTOCOL*)

Saves component at file path.

**Parameters**

- **file_path** (*str*) – Location to save file.
- **pickle_protocol** (*int*) – The pickle data stream format.

`transform`(*self*, *X*, *y=None*)

Computes the feature matrix for the input X using featuretools' dfs algorithm.

**Parameters**

- **X** (*pd.DataFrame or np.ndarray*) – The input training data to transform. Has shape [n_samples, n_features]

- **y** (*pd.Series, optional*) – Ignored.

**Returns** Feature matrix

**Return type** pd.DataFrame

**update_parameters**(*self*, *update_dict*, *reset_fit=True*)

Updates the parameter dictionary of the component.

**Parameters**

- **update_dict** (*dict*) – A dict of parameters to update.
- **reset_fit** (*bool, optional*) – If True, will set *_is_fitted* to False.

## log_transformer

Component that applies a log transformation to the target data.

### Module Contents

### Classes Summary

| | |
|---|---|
| *LogTransformer* | Applies a log transformation to the target data. |

### Contents

**class** evalml.pipelines.components.transformers.preprocessing.log_transformer.**LogTransformer**(*random_seed=*

Applies a log transformation to the target data.

**Attributes**

| | |
|---|---|
| **hyper-parameter_ranges** | {} |
| **modifies_features** | False |
| **modifies_target** | True |
| **name** | Log Transformer |
| **training_only** | False |

**Methods**

| | |
|---|---|
| *clone* | Constructs a new component with the same parameters and random state. |
| *default_parameters* | Returns the default parameters for this component. |
| *describe* | Describe a component and its parameters. |
| *fit* | Fits the LogTransformer. |
| *fit_transform* | Log transforms the target variable. |
| *inverse_transform* | Apply exponential to target data. |
| *load* | Loads component at file path. |
| *needs_fitting* | Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances. |
| *parameters* | Returns the parameters which were used to initialize the component. |
| *save* | Saves component at file path. |
| *transform* | Log transforms the target variable. |
| *update_parameters* | Updates the parameter dictionary of the component. |

**clone**(*self*)

Constructs a new component with the same parameters and random state.

**Returns** A new instance of this component with identical parameters and random state.

**default_parameters**(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

**Returns** Default parameters for this component.

**Return type** dict

**describe**(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

**Parameters**

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description as dictionary in the format {"name": name, "parameters": parameters}

**Returns** Returns dictionary if return_dict is True, else None.

**Return type** None or dict

**fit**(*self*, *X*, *y=None*)

Fits the LogTransformer.

**Parameters**

- **X** (*pd.DataFrame or np.ndarray*) – Ignored.
- **y** (*pd.Series, optional*) – Ignored.

**Returns** self

**fit_transform**(*self*, *X*, *y=None*)

Log transforms the target variable.

**Parameters**

- **X** (*pd.DataFrame, optional*) – Ignored.
- **y** (*pd.Series*) – Target variable to log transform.

**Returns**

**The input features are returned without modification. The target** variable y is log transformed.

**Return type** tuple of pd.DataFrame, pd.Series

**inverse_transform**(*self*, *y*)

Apply exponential to target data.

**Parameters** **y** (*pd.Series*) – Target variable.

**Returns** Target with exponential applied.

**Return type** pd.Series

**static load**(*file_path*)

Loads component at file path.

**Parameters** **file_path** (*str*) – Location to load file.

**Returns** ComponentBase object

**needs_fitting**(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need to be fit or whose fit methods do nothing.

**Returns** True.

**property parameters**(*self*)

Returns the parameters which were used to initialize the component.

**save**(*self*, *file_path*, *pickle_protocol=cloudpickle.DEFAULT_PROTOCOL*)

Saves component at file path.

**Parameters**

- **file_path** (*str*) – Location to save file.
- **pickle_protocol** (*int*) – The pickle data stream format.

**transform**(*self*, *X*, *y=None*)

Log transforms the target variable.

**Parameters**

- **X** (*pd.DataFrame, optional*) – Ignored.
- **y** (*pd.Series*) – Target data to log transform.

**Returns**

**The input features are returned without modification. The target** variable y is log transformed.

**Return type** tuple of pd.DataFrame, pd.Series

**update_parameters**(*self*, *update_dict*, *reset_fit=True*)

Updates the parameter dictionary of the component.

**Parameters**

- **update_dict** (*dict*) – A dict of parameters to update.
- **reset_fit** (*bool, optional*) – If True, will set *_is_fitted* to False.

## lsa

Transformer to calculate the Latent Semantic Analysis Values of text input.

## Module Contents

## Classes Summary

| | |
|---|---|
| *LSA* | Transformer to calculate the Latent Semantic Analysis Values of text input. |

## Contents

**class** evalml.pipelines.components.transformers.preprocessing.lsa.**LSA**(*random_seed=0*, ***kwargs*)

Transformer to calculate the Latent Semantic Analysis Values of text input.

**Parameters** **random_seed** (*int*) – Seed for the random number generator. Defaults to 0.

**Attributes**

| | |
|---|---|
| **hyper-parameter_ranges** | {} |
| **modifies_features** | True |
| **modifies_target** | False |
| **name** | LSA Transformer |
| **training_only** | False |

**Methods**

| | |
|---|---|
| *clone* | Constructs a new component with the same parameters and random state. |
| *default_parameters* | Returns the default parameters for this component. |
| *describe* | Describe a component and its parameters. |
| *fit* | Fits the input data. |
| *fit_transform* | Fits on X and transforms X. |
| *load* | Loads component at file path. |
| *needs_fitting* | Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances. |
| *parameters* | Returns the parameters which were used to initialize the component. |
| *save* | Saves component at file path. |
| *transform* | Transforms data X by applying the LSA pipeline. |
| *update_parameters* | Updates the parameter dictionary of the component. |

**clone**(*self*)

Constructs a new component with the same parameters and random state.

**Returns** A new instance of this component with identical parameters and random state.

**default_parameters**(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

**Returns** Default parameters for this component.

**Return type** dict

**describe**(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

**Parameters**

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description as dictionary in the format {"name": name, "parameters": parameters}

**Returns** Returns dictionary if return_dict is True, else None.

**Return type** None or dict

**fit**(*self*, *X*, *y=None*)

Fits the input data.

**Parameters**

- **X** (*pd.DataFrame*) – The data to transform.
- **y** (*pd.Series, optional*) – Ignored.

**Returns** self

**fit_transform**(*self*, *X*, *y=None*)

Fits on X and transforms X.

**Parameters**

- **X** (*pd.DataFrame*) – Data to fit and transform.

- **y** (*pd.Series*) – Target data.

**Returns** Transformed X.

**Return type** pd.DataFrame

**Raises** **MethodPropertyNotFoundError** – If transformer does not have a transform method or a component_obj that implements transform.

**static load**(*file_path*)

Loads component at file path.

**Parameters** **file_path** (*str*) – Location to load file.

**Returns** ComponentBase object

**needs_fitting**(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need to be fit or whose fit methods do nothing.

**Returns** True.

**property parameters**(*self*)

Returns the parameters which were used to initialize the component.

**save**(*self*, *file_path*, *pickle_protocol=cloudpickle.DEFAULT_PROTOCOL*)

Saves component at file path.

**Parameters**

- **file_path** (*str*) – Location to save file.
- **pickle_protocol** (*int*) – The pickle data stream format.

**transform**(*self*, *X*, *y=None*)

Transforms data X by applying the LSA pipeline.

**Parameters**

- **X** (*pd.DataFrame*) – The data to transform.
- **y** (*pd.Series, optional*) – Ignored.

**Returns**

**Transformed X. The original column is removed and replaced with two columns of the** format *LSA(original_column_name)[feature_number]*, where *feature_number* is 0 or 1.

**Return type** pd.DataFrame

**update_parameters**(*self*, *update_dict*, *reset_fit=True*)

Updates the parameter dictionary of the component.

**Parameters**

- **update_dict** (*dict*) – A dict of parameters to update.
- **reset_fit** (*bool, optional*) – If True, will set *_is_fitted* to False.

### natural_language_featurizer

Transformer that can automatically featurize text columns using featuretools' nlp_primitives.

### Module Contents

### Classes Summary

| | |
|---|---|
| *NaturalLanguageFeaturizer* | Transformer that can automatically featurize text columns using featuretools' nlp_primitives. |

### Contents

**class** evalml.pipelines.components.transformers.preprocessing.natural_language_featurizer.**NaturalLanguag**

Transformer that can automatically featurize text columns using featuretools' nlp_primitives.

Since models cannot handle non-numeric data, any text must be broken down into features that provide useful information about that text. This component splits each text column into several informative features: Diversity Score, Mean Characters per Word, Polarity Score, LSA (Latent Semantic Analysis), Number of Characters, and Number of Words. Calling transform on this component will replace any text columns in the given dataset with these numeric columns.

**Parameters** **random_seed** (*int*) – Seed for the random number generator. Defaults to 0.

**Attributes**

| | |
|---|---|
| **hyper-parameter_ranges** | {} |
| **modifies_features** | True |
| **modifies_target** | False |
| **name** | Natural Language Featurizer |
| **training_only** | False |

**Methods**

| | |
|---|---|
| *clone* | Constructs a new component with the same parameters and random state. |
| *default_parameters* | Returns the default parameters for this component. |
| *describe* | Describe a component and its parameters. |
| *fit* | Fits component to data. |
| *fit_transform* | Fits on X and transforms X. |
| *load* | Loads component at file path. |
| *needs_fitting* | Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances. |
| *parameters* | Returns the parameters which were used to initialize the component. |
| *save* | Saves component at file path. |
| *transform* | Transforms data X by creating new features using existing text columns. |
| *update_parameters* | Updates the parameter dictionary of the component. |

**clone**(*self*)

Constructs a new component with the same parameters and random state.

**Returns** A new instance of this component with identical parameters and random state.

**default_parameters**(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

**Returns** Default parameters for this component.

**Return type** dict

**describe**(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

**Parameters**

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description as dictionary in the format {"name": name, "parameters": parameters}

**Returns** Returns dictionary if return_dict is True, else None.

**Return type** None or dict

**fit**(*self*, *X*, *y=None*)

Fits component to data.

**Parameters**

- **X** (*pd.DataFrame or np.ndarray*) – The input training data of shape [n_samples, n_features]
- **y** (*pd.Series*) – The target training data of length [n_samples]

**Returns** self

**fit_transform**(*self*, *X*, *y=None*)

Fits on X and transforms X.

**Parameters**

- **X** (*pd.DataFrame*) – Data to fit and transform.
- **y** (*pd.Series*) – Target data.

**Returns** Transformed X.

**Return type** pd.DataFrame

**Raises** **MethodPropertyNotFoundError** – If transformer does not have a transform method or a component_obj that implements transform.

**static load**(*file_path*)

Loads component at file path.

**Parameters** **file_path** (*str*) – Location to load file.

**Returns** ComponentBase object

**needs_fitting**(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need to be fit or whose fit methods do nothing.

**Returns** True.

**property parameters**(*self*)

Returns the parameters which were used to initialize the component.

**save**(*self*, *file_path*, *pickle_protocol=cloudpickle.DEFAULT_PROTOCOL*)

Saves component at file path.

**Parameters**

- **file_path** (*str*) – Location to save file.
- **pickle_protocol** (*int*) – The pickle data stream format.

**transform**(*self*, *X*, *y=None*)

Transforms data X by creating new features using existing text columns.

**Parameters**

- **X** (*pd.DataFrame*) – The data to transform.
- **y** (*pd.Series, optional*) – Ignored.

**Returns** Transformed X

**Return type** pd.DataFrame

**update_parameters**(*self*, *update_dict*, *reset_fit=True*)

Updates the parameter dictionary of the component.

**Parameters**

- **update_dict** (*dict*) – A dict of parameters to update.
- **reset_fit** (*bool, optional*) – If True, will set *_is_fitted* to False.

### polynomial_decomposer

Component that removes trends from time series by fitting a polynomial to the data.

### Module Contents

### Classes Summary

| | |
|---|---|
| *PolynomialDecomposer* | Removes trends and seasonality from time series by fitting a polynomial and moving average to the data. |

### Contents

**class** evalml.pipelines.components.transformers.preprocessing.polynomial_decomposer.**PolynomialDecomposer**

Removes trends and seasonality from time series by fitting a polynomial and moving average to the data.

**Scikit-learn's PolynomialForecaster is used to generate the additive trend portion of the target data. A polynomial** will be fit to the data during fit. That additive polynomial trend will be removed during fit so that statsmodel's seasonal_decompose can determine the addititve seasonality of the data by using rolling averages over the series' inferred periodicity.

For example, daily time series data will generate rolling averages over the first week of data, normalize out the mean and return those 7 averages repeated over the entire length of the given series. Those seven averages, repeated as many times as necessary to match the length of the given target data, will be used as the seasonal signal of the data.

**Parameters**

- **time_index** (*str*) – Specifies the name of the column in X that provides the datetime objects. Defaults to None.

- **degree** (*int*) – Degree for the polynomial. If 1, linear model is fit to the data. If 2, quadratic model is fit, etc. Defaults to 1.
- **period** (*int*) – The number of entries in the time series data that corresponds to one period of a cyclic signal. For instance, if data is known to possess a weekly seasonal signal, and if the data is daily data, period should be 7. For daily data with a yearly seasonal signal, period should be 365. Defaults to -1, which uses the statsmodels libarary's freq_to_period function. https://github.com/statsmodels/statsmodels/blob/main/statsmodels/tsa/tsatools.py
- **random_seed** (*int*) – Seed for the random number generator. Defaults to 0.

**Attributes**

| | |
|---|---|
| **hyper-parameter_ranges** | { "degree": Integer(1, 3)} |
| **invalid_frequencies** | [] |
| **modifies_features** | False |
| **modifies_target** | True |
| **name** | Polynomial Decomposer |
| **needs_fitting** | True |
| **training_only** | False |

**Methods**

| | |
|---|---|
| *clone* | Constructs a new component with the same parameters and random state. |
| *default_parameters* | Returns the default parameters for this component. |
| *describe* | Describe a component and its parameters. |
| *determine_periodicity* | Function that uses autocorrelative methods to determine the likely most signficant period of the seasonal signal. |
| *fit* | Fits the PolynomialDecomposer and determine the seasonal signal. |
| *fit_transform* | Removes fitted trend and seasonality from target variable. |
| *get_trend_dataframe* | Return a list of dataframes with 4 columns: signal, trend, seasonality, residual. |
| *inverse_transform* | Adds back fitted trend and seasonality to target variable. |
| *is_freq_valid* | Determines if the given string represents a valid frequency for this decomposer. |
| *load* | Loads component at file path. |
| *parameters* | Returns the parameters which were used to initialize the component. |
| *plot_decomposition* | Plots the decomposition of the target signal. |
| *save* | Saves component at file path. |
| *set_period* | Function to set the component's seasonal period based on the target's seasonality. |
| *transform* | Transforms the target data by removing the polynomial trend and rolling average seasonality. |
| *update_parameters* | Updates the parameter dictionary of the component. |

**clone**(*self*)

Constructs a new component with the same parameters and random state.

**Returns** A new instance of this component with identical parameters and random state.

**default_parameters**(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

**Returns** Default parameters for this component.

**Return type** dict

**describe**(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

**Parameters**

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description as dictionary in the format {"name": name, "parameters": parameters}

**Returns** Returns dictionary if return_dict is True, else None.

**Return type** None or dict

**classmethod determine_periodicity**(*cls*, *X: pandas.DataFrame*, *y: pandas.Series*, *acf_threshold: float = 0.01*, *rel_max_order: int = 5*)

Function that uses autocorrelative methods to determine the likely most signficant period of the seasonal signal.

**Parameters**

- **X** (*pandas.DataFrame*) – The feature data of the time series problem.
- **y** (*pandas.Series*) – The target data of a time series problem.
- **acf_threshold** (*float*) – The threshold for the autocorrelation function to determine the period. Any values below the threshold are considered to be 0 and will not be considered for the period. Defaults to 0.01.
- **rel_max_order** (*int*) – The order of the relative maximum to determine the period. Defaults to 5.

**Returns**

**The integer number of entries in time series data over which the seasonal part of the target data** repeats. If the time series data is in days, then this is the number of days that it takes the target's seasonal signal to repeat. Note: the target data can contain multiple seasonal signals. This function will only return the stronger. E.g. if the target has both weekly and yearly seasonality, the function may return either "7" or "365", depending on which seasonality is more strongly autocorrelated. If no period is detected, returns None.

**Return type** int

**fit**(*self*, *X: pandas.DataFrame*, *y: pandas.Series = None*) → *PolynomialDecomposer*

Fits the PolynomialDecomposer and determine the seasonal signal.

Currently only fits the polynomial detrender. The seasonality is determined by removing the trend from the signal and using statsmodels' seasonal_decompose(). Both the trend and seasonality are currently assumed to be additive.

**Parameters**

- **X** (*pd.DataFrame, optional*) – Conditionally used to build datetime index.
- **y** (*pd.Series*) – Target variable to detrend and deseasonalize.

**Returns** self

**Raises**

- **NotImplementedError** – If the input data has a frequency of "month-begin". This isn't supported by statsmodels decompose as the freqstr "MS" is misinterpreted as milliseconds.
- **ValueError** – If y is None.
- **ValueError** – If target data doesn't have DatetimeIndex AND no Datetime features in features data

**fit_transform**(*self*, *X: pandas.DataFrame*, *y: pandas.Series = None*) → tuple[pandas.DataFrame, pandas.Series]

Removes fitted trend and seasonality from target variable.

**Parameters**

- **X** (*pd.DataFrame, optional*) – Ignored.
- **y** (*pd.Series*) – Target variable to detrend and deseasonalize.

**Returns**

**The first element are the input features returned without modification.** The second element is the target variable y with the fitted trend removed.

**Return type** tuple of pd.DataFrame, pd.Series

`get_trend_dataframe`(*self*, *X: pandas.DataFrame*, *y: pandas.Series*) → list[pandas.DataFrame]

Return a list of dataframes with 4 columns: signal, trend, seasonality, residual.

Scikit-learn's PolynomialForecaster is used to generate the trend portion of the target data. statsmodel's seasonal_decompose is used to generate the seasonality of the data.

**Parameters**

- **X** (*pd.DataFrame*) – Input data with time series data in index.
- **y** (*pd.Series or pd.DataFrame*) – Target variable data provided as a Series for univariate problems or a DataFrame for multivariate problems.

**Returns**

**Each DataFrame contains the columns "signal", "trend", "seasonality" and "residual,"** with the latter 3 column values being the decomposed elements of the target data. The "signal" column is simply the input target signal but reindexed with a datetime index to match the input features.

**Return type** list of pd.DataFrame

**Raises**

- **TypeError** – If X does not have time-series data in the index.
- **ValueError** – If time series index of X does not have an inferred frequency.
- **ValueError** – If the forecaster associated with the detrender has not been fit yet.
- **TypeError** – If y is not provided as a pandas Series or DataFrame.

`inverse_transform`(*self*, *y_t: pandas.Series*) → tuple[pandas.DataFrame, pandas.Series]

Adds back fitted trend and seasonality to target variable.

The polynomial trend is added back into the signal, calling the detrender's inverse_transform(). Then, the seasonality is projected forward to and added back into the signal.

**Parameters** **y_t** (*pd.Series*) – Target variable.

**Returns**

**The first element are the input features returned without modification.** The second element is the target variable y with the trend and seasonality added back in.

**Return type** tuple of pd.DataFrame, pd.Series

**Raises** **ValueError** – If y is None.

`classmethod is_freq_valid`(*cls*, *freq: str*)

Determines if the given string represents a valid frequency for this decomposer.

**Parameters** **freq** (*str*) – A frequency to validate. See the pandas docs at https://pandas.pydata.org/pandas-docs/stable/user_guide/timeseries.html#offset-aliases for options.

**Returns** boolean representing whether the frequency is valid or not.

`static load`(*file_path*)

Loads component at file path.

**Parameters** **file_path** (*str*) – Location to load file.

**Returns** ComponentBase object

**property parameters**(*self*)

Returns the parameters which were used to initialize the component.

**plot_decomposition**(*self*, *X: pandas.DataFrame*, *y: pandas.Series*, *show: bool = False*) → tuple[matplotlib.pyplot.Figure, list]

Plots the decomposition of the target signal.

**Parameters**

- **X** (*pd.DataFrame*) – Input data with time series data in index.
- **y** (*pd.Series or pd.DataFrame*) – Target variable data provided as a Series for univariate problems or a DataFrame for multivariate problems.
- **show** (*bool*) – Whether to display the plot or not. Defaults to False.

**Returns**

**The figure and axes that have the decompositions** plotted on them

**Return type** matplotlib.pyplot.Figure, list[matplotlib.pyplot.Axes]

**save**(*self*, *file_path*, *pickle_protocol=cloudpickle.DEFAULT_PROTOCOL*)

Saves component at file path.

**Parameters**

- **file_path** (*str*) – Location to save file.
- **pickle_protocol** (*int*) – The pickle data stream format.

**set_period**(*self*, *X: pandas.DataFrame*, *y: pandas.Series*, *acf_threshold: float = 0.01*, *rel_max_order: int = 5*)

Function to set the component's seasonal period based on the target's seasonality.

**Parameters**

- **X** (*pandas.DataFrame*) – The feature data of the time series problem.
- **y** (*pandas.Series*) – The target data of a time series problem.
- **acf_threshold** (*float*) – The threshold for the autocorrelation function to determine the period. Any values below the threshold are considered to be 0 and will not be considered for the period. Defaults to 0.01.
- **rel_max_order** (*int*) – The order of the relative maximum to determine the period. Defaults to 5.

**transform**(*self*, *X: pandas.DataFrame*, *y: pandas.Series = None*) → tuple[pandas.DataFrame, pandas.Series]

Transforms the target data by removing the polynomial trend and rolling average seasonality.

Applies the fit polynomial detrender to the target data, removing the additive polynomial trend. Then, utilizes the first period's worth of seasonal data determined in the .fit() function to extrapolate the seasonal signal of the data to be transformed. This seasonal signal is also assumed to be additive and is removed.

**Parameters**

- **X** (*pd.DataFrame, optional*) – Conditionally used to build datetime index.
- **y** (*pd.Series*) – Target variable to detrend and deseasonalize.

**Returns**

**The input features are returned without modification. The target** variable y is detrended and deseasonalized.

**Return type** tuple of pd.DataFrame, pd.Series

**Raises** `ValueError` – If target data doesn't have DatetimeIndex AND no Datetime features in features data

`update_parameters`(*self*, *update_dict*, *reset_fit=True*)

Updates the parameter dictionary of the component.

**Parameters**

- `update_dict` (*dict*) – A dict of parameters to update.
- `reset_fit` (*bool, optional*) – If True, will set *_is_fitted* to False.

## replace_nullable_types

Transformer to replace features with the new nullable dtypes with a dtype that is compatible in EvalML.

### Module Contents

### Classes Summary

| | |
|---|---|
| *ReplaceNullableTypes* | Transformer to replace features with the new nullable dtypes with a dtype that is compatible in EvalML. |

### Contents

**class** `evalml.pipelines.components.transformers.preprocessing.replace_nullable_types.`**`ReplaceNullableTypes`**

Transformer to replace features with the new nullable dtypes with a dtype that is compatible in EvalML.

**Attributes**

| | |
|---|---|
| **hyper-parameter_ranges** | None |
| **modifies_features** | True |
| **modifies_target** | {} |
| **name** | Replace Nullable Types Transformer |
| **training_only** | False |

**Methods**

| | |
|---|---|
| *clone* | Constructs a new component with the same parameters and random state. |
| *default_parameters* | Returns the default parameters for this component. |
| *describe* | Describe a component and its parameters. |
| *fit* | Fits component to data. |
| *fit_transform* | Substitutes non-nullable types for the new pandas nullable types in the data and target data. |
| *load* | Loads component at file path. |
| *needs_fitting* | Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances. |
| *parameters* | Returns the parameters which were used to initialize the component. |
| *save* | Saves component at file path. |
| *transform* | Transforms data by replacing columns that contain nullable types with the appropriate replacement type. |
| *update_parameters* | Updates the parameter dictionary of the component. |

**clone**(*self*)

Constructs a new component with the same parameters and random state.

**Returns** A new instance of this component with identical parameters and random state.

**default_parameters**(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

**Returns** Default parameters for this component.

**Return type** dict

**describe**(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

**Parameters**

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description as dictionary in the format {"name": name, "parameters": parameters}

**Returns** Returns dictionary if return_dict is True, else None.

**Return type** None or dict

**fit**(*self*, *X*, *y=None*)

Fits component to data.

**Parameters**

- **X** (*pd.DataFrame*) – The input training data of shape [n_samples, n_features].
- **y** (*pd.Series, optional*) – The target training data of length [n_samples].

**Returns** self

**fit_transform**(*self*, *X*, *y=None*)

Substitutes non-nullable types for the new pandas nullable types in the data and target data.

**Parameters**

- **X** (*pd.DataFrame, optional*) – Input features.
- **y** (*pd.Series*) – Target data.

**Returns** The input features and target data with the non-nullable types set.

**Return type** tuple of pd.DataFrame, pd.Series

**static load**(*file_path*)

Loads component at file path.

**Parameters** **file_path** (*str*) – Location to load file.

**Returns** ComponentBase object

**needs_fitting**(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need to be fit or whose fit methods do nothing.

**Returns** True.

**property parameters**(*self*)

Returns the parameters which were used to initialize the component.

**save**(*self*, *file_path*, *pickle_protocol=cloudpickle.DEFAULT_PROTOCOL*)

Saves component at file path.

**Parameters**

- **file_path** (*str*) – Location to save file.
- **pickle_protocol** (*int*) – The pickle data stream format.

**transform**(*self*, *X*, *y=None*)

Transforms data by replacing columns that contain nullable types with the appropriate replacement type.

"float64" for nullable integers and "category" for nullable booleans.

**Parameters**

- **X** (*pd.DataFrame*) – Data to transform
- **y** (*pd.Series, optional*) – Target data to transform

**Returns** Transformed X pd.Series: Transformed y

**Return type** pd.DataFrame

**update_parameters**(*self*, *update_dict*, *reset_fit=True*)

Updates the parameter dictionary of the component.

**Parameters**

- **update_dict** (*dict*) – A dict of parameters to update.
- **reset_fit** (*bool, optional*) – If True, will set *_is_fitted* to False.

### stl_decomposer

Component that removes trends and seasonality from time series using STL.

### Module Contents

### Classes Summary

| | |
|---|---|
| *STLDecomposer* | Removes trends and seasonality from time series using the STL algorithm. |

### Contents

**class** evalml.pipelines.components.transformers.preprocessing.stl_decomposer.**STLDecomposer**(*time_index: str = None, degree: int = 1, period: int = None, seasonal_smoother: int = 7, random_seed: int = 0, **kwargs*)

Removes trends and seasonality from time series using the STL algorithm.

https://www.statsmodels.org/dev/generated/statsmodels.tsa.seasonal.STL.html

**Parameters**

- **time_index** (*str*) – Specifies the name of the column in X that provides the datetime objects. Defaults to None.
- **degree** (*int*) – Not currently used. STL 3x "degree-like" values. None are able to be set at this time. Defaults to 1.
- **period** (*int*) – The number of entries in the time series data that corresponds to one period of a cyclic signal. For instance, if data is known to possess a weekly seasonal signal, and

if the data is daily data, the period should likely be 7. For daily data with a yearly seasonal signal, the period should likely be 365. If None, statsmodels will infer the period based on the frequency. Defaults to None.

- **seasonal_smoother** (*int*) – The length of the seasonal smoother used by the underlying STL algorithm. For compatibility, must be odd. If an even number is provided, the next, highest odd number will be used. Defaults to 7.
- **random_seed** (*int*) – Seed for the random number generator. Defaults to 0.

**Attributes**

| | |
|---|---|
| **hyper-parameter_ranges** | None |
| **invalid_frequencies** | [] |
| **modifies_features** | False |
| **modifies_target** | True |
| **name** | STL Decomposer |
| **needs_fitting** | True |
| **training_only** | False |

**Methods**

| | |
|---|---|
| clone | Constructs a new component with the same parameters and random state. |
| default_parameters | Returns the default parameters for this component. |
| describe | Describe a component and its parameters. |
| determine_periodicity | Function that uses autocorrelative methods to determine the likely most signficant period of the seasonal signal. |
| fit | Fits the STLDecomposer and determine the seasonal signal. |
| fit_transform | Removes fitted trend and seasonality from target variable. |
| get_trend_dataframe | Return a list of dataframes with 4 columns: signal, trend, seasonality, residual. |
| get_trend_prediction_intervals | Calculate the prediction intervals for the trend data. |
| inverse_transform | Adds back fitted trend and seasonality to target variable. |
| is_freq_valid | Determines if the given string represents a valid frequency for this decomposer. |
| load | Loads component at file path. |
| parameters | Returns the parameters which were used to initialize the component. |
| plot_decomposition | Plots the decomposition of the target signal. |
| save | Saves component at file path. |
| set_period | Function to set the component's seasonal period based on the target's seasonality. |
| transform | Transforms the target data by removing the STL trend and seasonality. |
| update_parameters | Updates the parameter dictionary of the component. |

**clone**(*self*)

Constructs a new component with the same parameters and random state.

**Returns** A new instance of this component with identical parameters and random state.

**default_parameters**(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

**Returns** Default parameters for this component.

**Return type** dict

**describe**(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

**Parameters**

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description as dictionary in the format {"name": name, "parameters": parameters}

**Returns** Returns dictionary if return_dict is True, else None.

**Return type** None or dict

**classmethod determine_periodicity**(*cls*, *X: pandas.DataFrame*, *y: pandas.Series*, *acf_threshold: float = 0.01*, *rel_max_order: int = 5*)

Function that uses autocorrelative methods to determine the likely most signficant period of the seasonal signal.

**Parameters**

- **X** (*pandas.DataFrame*) – The feature data of the time series problem.
- **y** (*pandas.Series*) – The target data of a time series problem.
- **acf_threshold** (*float*) – The threshold for the autocorrelation function to determine the period. Any values below the threshold are considered to be 0 and will not be considered for the period. Defaults to 0.01.
- **rel_max_order** (*int*) – The order of the relative maximum to determine the period. Defaults to 5.

**Returns**

**The integer number of entries in time series data over which the seasonal part of the target data** repeats. If the time series data is in days, then this is the number of days that it takes the target's seasonal signal to repeat. Note: the target data can contain multiple seasonal signals. This function will only return the stronger. E.g. if the target has both weekly and yearly seasonality, the function may return either "7" or "365", depending on which seasonality is more strongly autocorrelated. If no period is detected, returns None.

**Return type** int

**fit**(*self*, *X: pandas.DataFrame*, *y: pandas.Series = None*) → *STLDecomposer*

Fits the STLDecomposer and determine the seasonal signal.

Instantiates a statsmodels STL decompose object with the component's stored parameters and fits it. Since the statsmodels object does not fit the sklearn api, it is not saved during __init__() in _component_obj and will be re-instantiated each time fit is called.

To emulate the sklearn API, when the STL decomposer is fit, the full seasonal component, a single period sample of the seasonal component, the full trend-cycle component and the residual are saved.

y(t) = S(t) + T(t) + R(t)

**Parameters**

- **X** (*pd.DataFrame, optional*) – Conditionally used to build datetime index.
- **y** (*pd.Series*) – Target variable to detrend and deseasonalize.

**Returns** self

**Raises**

- **ValueError** – If y is None.
- **ValueError** – If target data doesn't have DatetimeIndex AND no Datetime features in features data

**fit_transform**(*self*, *X: pandas.DataFrame*, *y: pandas.Series = None*) → tuple[pandas.DataFrame, pandas.Series]

Removes fitted trend and seasonality from target variable.

**Parameters**

- **X** (*pd.DataFrame, optional*) – Ignored.
- **y** (*pd.Series*) – Target variable to detrend and deseasonalize.

**Returns**

**The first element are the input features returned without modification.** The second element is the target variable y with the fitted trend removed.

**Return type** tuple of pd.DataFrame, pd.Series

`get_trend_dataframe`(*self*, *X*, *y*)

Return a list of dataframes with 4 columns: signal, trend, seasonality, residual.

**Parameters**

- **X** (*pd.DataFrame*) – Input data with time series data in index.
- **y** (*pd.Series or pd.DataFrame*) – Target variable data provided as a Series for univariate problems or a DataFrame for multivariate problems.

**Returns**

**Each DataFrame contains the columns "signal", "trend", "seasonality" and "residual,"** with the latter 3 column values being the decomposed elements of the target data. The "signal" column is simply the input target signal but reindexed with a datetime index to match the input features.

**Return type** list of pd.DataFrame

**Raises**

- **TypeError** – If X does not have time-series data in the index.
- **ValueError** – If time series index of X does not have an inferred frequency.
- **ValueError** – If the forecaster associated with the detrender has not been fit yet.
- **TypeError** – If y is not provided as a pandas Series or DataFrame.

`get_trend_prediction_intervals`(*self*, *y*, *coverage=None*)

Calculate the prediction intervals for the trend data.

**Parameters**

- **y** (*pd.Series*) – Target data.
- **coverage** (*list[float]*) – A list of floats between the values 0 and 1 that the upper and lower bounds of the prediction interval should be calculated for.

**Returns** Prediction intervals, keys are in the format {coverage}_lower or {coverage}_upper.

**Return type** dict of pd.Series

`inverse_transform`(*self*, *y_t: pandas.Series*) → tuple[pandas.DataFrame, pandas.Series]

Adds back fitted trend and seasonality to target variable.

The STL trend is projected to cover the entire requested target range, then added back into the signal. Then, the seasonality is projected forward to and added back into the signal.

**Parameters** **y_t** (*pd.Series*) – Target variable.

**Returns**

**The first element are the input features returned without modification.** The second element is the target variable y with the trend and seasonality added back in.

**Return type** tuple of pd.DataFrame, pd.Series

**Raises** **ValueError** – If y is None.

**classmethod is_freq_valid**(*cls*, *freq: str*)

Determines if the given string represents a valid frequency for this decomposer.

**Parameters** **freq** (*str*) – A frequency to validate. See the pandas docs at https://pandas.pydata.org/pandas-docs/stable/user_guide/timeseries.html#offset-aliases for options.

**Returns** boolean representing whether the frequency is valid or not.

**static load**(*file_path*)

Loads component at file path.

**Parameters** **file_path** (*str*) – Location to load file.

**Returns** ComponentBase object

**property parameters**(*self*)

Returns the parameters which were used to initialize the component.

**plot_decomposition**(*self*, *X: pandas.DataFrame*, *y: pandas.Series*, *show: bool = False*) → tuple[matplotlib.pyplot.Figure, list]

Plots the decomposition of the target signal.

**Parameters**

- **X** (*pd.DataFrame*) – Input data with time series data in index.
- **y** (*pd.Series or pd.DataFrame*) – Target variable data provided as a Series for univariate problems or a DataFrame for multivariate problems.
- **show** (*bool*) – Whether to display the plot or not. Defaults to False.

**Returns**

**The figure and axes that have the decompositions** plotted on them

**Return type** matplotlib.pyplot.Figure, list[matplotlib.pyplot.Axes]

**save**(*self*, *file_path*, *pickle_protocol=cloudpickle.DEFAULT_PROTOCOL*)

Saves component at file path.

**Parameters**

- **file_path** (*str*) – Location to save file.
- **pickle_protocol** (*int*) – The pickle data stream format.

**set_period**(*self*, *X: pandas.DataFrame*, *y: pandas.Series*, *acf_threshold: float = 0.01*, *rel_max_order: int = 5*)

Function to set the component's seasonal period based on the target's seasonality.

**Parameters**

- **X** (*pandas.DataFrame*) – The feature data of the time series problem.
- **y** (*pandas.Series*) – The target data of a time series problem.
- **acf_threshold** (*float*) – The threshold for the autocorrelation function to determine the period. Any values below the threshold are considered to be 0 and will not be considered for the period. Defaults to 0.01.
- **rel_max_order** (*int*) – The order of the relative maximum to determine the period. Defaults to 5.

**transform**(*self*, *X: pandas.DataFrame*, *y: pandas.Series = None*) → tuple[pandas.DataFrame, pandas.Series]

Transforms the target data by removing the STL trend and seasonality.

Uses an ARIMA model to project forward the addititve trend and removes it. Then, utilizes the first period's worth of seasonal data determined in the .fit() function to extrapolate the seasonal signal of the data to be transformed. This seasonal signal is also assumed to be additive and is removed.

**Parameters**

- **X** (*pd.DataFrame, optional*) – Conditionally used to build datetime index.
- **y** (*pd.Series*) – Target variable to detrend and deseasonalize.

**Returns**

**The input features are returned without modification. The target** variable y is detrended and deseasonalized.

**Return type** tuple of pd.DataFrame, pd.Series

**Raises** **ValueError** – If target data doesn't have DatetimeIndex AND no Datetime features in features data

**update_parameters**(*self*, *update_dict*, *reset_fit=True*)

Updates the parameter dictionary of the component.

**Parameters**

- **update_dict** (*dict*) – A dict of parameters to update.
- **reset_fit** (*bool, optional*) – If True, will set *_is_fitted* to False.

## text_transformer

Base class for all transformers working with text features.

## Module Contents

## Classes Summary

| | |
|---|---|
| *TextTransformer* | Base class for all transformers working with text features. |

## Contents

**class** evalml.pipelines.components.transformers.preprocessing.text_transformer.**TextTransformer**(*component_ran-dom_seed=0 **kwargs*)

Base class for all transformers working with text features.

**Parameters**

- **component_obj** (*obj*) – Third-party objects useful in component implementation. Defaults to None.

- **random_seed** (*int*) – Seed for the random number generator. Defaults to 0.

**Attributes**

| | |
|---|---|
| **modifies_features** | True |
| **modifies_target** | False |
| **training_only** | False |

**Methods**

| | |
|---|---|
| *clone* | Constructs a new component with the same parameters and random state. |
| *default_parameters* | Returns the default parameters for this component. |
| *describe* | Describe a component and its parameters. |
| *fit* | Fits component to data. |
| *fit_transform* | Fits on X and transforms X. |
| *load* | Loads component at file path. |
| *name* | Returns string name of this component. |
| *needs_fitting* | Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances. |
| *parameters* | Returns the parameters which were used to initialize the component. |
| *save* | Saves component at file path. |
| *transform* | Transforms data X. |
| *update_parameters* | Updates the parameter dictionary of the component. |

**clone**(*self*)

Constructs a new component with the same parameters and random state.

**Returns** A new instance of this component with identical parameters and random state.

**default_parameters**(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

**Returns** Default parameters for this component.

**Return type** dict

**describe**(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

**Parameters**

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description as dictionary in the format {"name": name, "parameters": parameters}

**Returns** Returns dictionary if return_dict is True, else None.

**Return type** None or dict

**fit**(*self*, *X*, *y=None*)

Fits component to data.

**Parameters**

- **X** (*pd.DataFrame*) – The input training data of shape [n_samples, n_features]
- **y** (*pd.Series, optional*) – The target training data of length [n_samples]

**Returns** self

**Raises** **MethodPropertyNotFoundError** – If component does not have a fit method or a component_obj that implements fit.

**fit_transform**(*self*, *X*, *y=None*)

Fits on X and transforms X.

**Parameters**

- **X** (*pd.DataFrame*) – Data to fit and transform.
- **y** (*pd.Series*) – Target data.

**Returns** Transformed X.

**Return type** pd.DataFrame

**Raises** **MethodPropertyNotFoundError** – If transformer does not have a transform method or a component_obj that implements transform.

**static load**(*file_path*)

Loads component at file path.

**Parameters** **file_path** (*str*) – Location to load file.

**Returns** ComponentBase object

**property name**(*cls*)

Returns string name of this component.

**needs_fitting**(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need to be fit or whose fit methods do nothing.

**Returns** True.

**property parameters**(*self*)

Returns the parameters which were used to initialize the component.

**save**(*self*, *file_path*, *pickle_protocol=cloudpickle.DEFAULT_PROTOCOL*)

Saves component at file path.

**Parameters**

- **file_path** (*str*) – Location to save file.
- **pickle_protocol** (*int*) – The pickle data stream format.

**abstract transform**(*self*, *X*, *y=None*)

Transforms data X.

**Parameters**

- **X** (*pd.DataFrame*) – Data to transform.

- **y** (*pd.Series, optional*) – Target data.

**Returns** Transformed X

**Return type** pd.DataFrame

**Raises** **MethodPropertyNotFoundError** – If transformer does not have a transform method or a component_obj that implements transform.

**update_parameters**(*self*, *update_dict*, *reset_fit=True*)

Updates the parameter dictionary of the component.

**Parameters**

- **update_dict** (*dict*) – A dict of parameters to update.
- **reset_fit** (*bool, optional*) – If True, will set *_is_fitted* to False.

### time_series_featurizer

Transformer that delays input features and target variable for time series problems.

### Module Contents

### Classes Summary

| | |
|---|---|
| *TimeSeriesFeaturizer* | Transformer that delays input features and target variable for time series problems. |

### Contents

**class** evalml.pipelines.components.transformers.preprocessing.time_series_featurizer.**TimeSeriesFeaturizer**

Transformer that delays input features and target variable for time series problems.

This component uses an algorithm based on the autocorrelation values of the target variable to determine which lags to select from the set of all possible lags.

The algorithm is based on the idea that the local maxima of the autocorrelation function indicate the lags that have the most impact on the present time.

The algorithm computes the autocorrelation values and finds the local maxima, called "peaks", that are significant at the given conf_level. Since lags in the range [0, 10] tend to be predictive but not local maxima, the union of the peaks is taken with the significant lags in the range [0, 10]. At the end, only selected lags in the range [0, max_delay] are used.

Parametrizing the algorithm by conf_level lets the AutoMLAlgorithm tune the set of lags chosen so that the chances of finding a good set of lags is higher.

Using conf_level value of 1 selects all possible lags.

**Parameters**

- **time_index** (*str*) – Name of the column containing the datetime information used to order the data. Ignored.
- **max_delay** (*int*) – Maximum number of time units to delay each feature. Defaults to 2.
- **forecast_horizon** (*int*) – The number of time periods the pipeline is expected to forecast.
- **conf_level** (*float*) – Float in range (0, 1] that determines the confidence interval size used to select which lags to compute from the set of [1, max_delay]. A delay of 1 will always be computed. If 1, selects all possible lags in the set of [1, max_delay], inclusive.
- **rolling_window_size** (*float*) – Float in range (0, 1] that determines the size of the window used for rolling features. Size is computed as rolling_window_size * max_delay.
- **delay_features** (*bool*) – Whether to delay the input features. Defaults to True.
- **delay_target** (*bool*) – Whether to delay the target. Defaults to True.
- **gap** (*int*) – The number of time units between when the features are collected and when the target is collected. For example, if you are predicting the next time step's target, gap=1. This is only needed because when gap=0, we need to be sure to start the lagging of the target variable at 1. Defaults to 1.
- **random_seed** (*int*) – Seed for the random number generator. This transformer performs the same regardless of the random seed provided.

**Attributes**

| | |
|---|---|
| **hyper-parameter_ranges** | Real(0.001, 1.0), "rolling_window_size": Real(0.001, 1.0)}:type: {"conf_level" |
| **modifies_features** | True |
| **modifies_target** | False |
| **name** | Time Series Featurizer |
| **needs_fitting** | True |
| **target_colname_prefix** | target_delay_{} |
| **training_only** | False |

**Methods**

| | |
|---|---|
| *clone* | Constructs a new component with the same parameters and random state. |
| *default_parameters* | Returns the default parameters for this component. |
| *describe* | Describe a component and its parameters. |
| *fit* | Fits the DelayFeatureTransformer. |
| *fit_transform* | Fit the component and transform the input data. |
| *load* | Loads component at file path. |
| *parameters* | Returns the parameters which were used to initialize the component. |
| *save* | Saves component at file path. |
| *transform* | Computes the delayed values and rolling means for X and y. |
| *update_parameters* | Updates the parameter dictionary of the component. |

**clone**(*self*)

Constructs a new component with the same parameters and random state.

**Returns** A new instance of this component with identical parameters and random state.

**default_parameters**(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

**Returns** Default parameters for this component.

**Return type** dict

**describe**(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

**Parameters**

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description as dictionary in the format {"name": name, "parameters": parameters}

**Returns** Returns dictionary if return_dict is True, else None.

**Return type** None or dict

**fit**(*self*, *X*, *y=None*)

Fits the DelayFeatureTransformer.

**Parameters**

- **X** (*pd.DataFrame or np.ndarray*) – The input training data of shape [n_samples, n_features]
- **y** (*pd.Series, optional*) – The target training data of length [n_samples]

**Returns** self

**Raises** **ValueError** – if self.time_index is None

**fit_transform**(*self*, *X*, *y=None*)

Fit the component and transform the input data.

**Parameters**

- **X** (*pd.DataFrame*) – Data to transform.
- **y** (*pd.Series, or None*) – Target.

**Returns** Transformed X.

**Return type** pd.DataFrame

**static load**(*file_path*)

Loads component at file path.

**Parameters** **file_path** (*str*) – Location to load file.

**Returns** ComponentBase object

**property parameters**(*self*)

Returns the parameters which were used to initialize the component.

**save**(*self*, *file_path*, *pickle_protocol=cloudpickle.DEFAULT_PROTOCOL*)

Saves component at file path.

**Parameters**

- **file_path** (*str*) – Location to save file.
- **pickle_protocol** (*int*) – The pickle data stream format.

**transform**(*self*, *X*, *y=None*)

Computes the delayed values and rolling means for X and y.

The chosen delays are determined by the autocorrelation function of the target variable. See the class docstring for more information on how they are chosen. If y is None, all possible lags are chosen.

If y is not None, it will also compute the delayed values for the target variable.

The rolling means for all numeric features in X and y, if y is numeric, are also returned.

**Parameters**

- **X** (*pd.DataFrame or None*) – Data to transform. None is expected when only the target variable is being used.
- **y** (*pd.Series, or None*) – Target.

**Returns** Transformed X. No original features are returned.

**Return type** pd.DataFrame

**update_parameters**(*self*, *update_dict*, *reset_fit=True*)

Updates the parameter dictionary of the component.

**Parameters**

- **update_dict** (*dict*) – A dict of parameters to update.
- **reset_fit** (*bool, optional*) – If True, will set *_is_fitted* to False.

### time_series_regularizer

Transformer that regularizes a dataset with an uninferrable offset frequency for time series problems.

### Module Contents

### Classes Summary

| | |
|---|---|
| *TimeSeriesRegularizer* | Transformer that regularizes an inconsistently spaced datetime column. |

### Contents

**class** evalml.pipelines.components.transformers.preprocessing.time_series_regularizer.**TimeSeriesRegulari**

Transformer that regularizes an inconsistently spaced datetime column.

If X is passed in to fit/transform, the column *time_index* will be checked for an inferrable offset frequency. If the *time_index* column is perfectly inferrable then this Transformer will do nothing and return the original X and y.

If X does not have a perfectly inferrable frequency but one can be estimated, then X and y will be reformatted based on the estimated frequency for *time_index*. In the original X and y passed: - Missing datetime values will be added and will have their corresponding columns in X and y set to None. - Duplicate datetime values will be dropped. - Extra datetime values will be dropped. - If it can be determined that a duplicate or extra value is misaligned, then it will be repositioned to take the place of a missing value.

This Transformer should be used before the *TimeSeriesImputer* in order to impute the missing values that were added to X and y (if passed).

**Parameters**

- **time_index** (*string*) – Name of the column containing the datetime information used to order the data, required. Defaults to None.
- **frequency_payload** (*tuple*) – Payload returned from Woodwork's infer_frequency function where debug is True. Defaults to None.
- **window_length** (*int*) – The size of the rolling window over which inference is conducted to determine the prevalence of uninferrable frequencies.
- **5.** (*Lower values make this component more sensitive to recognizing numerous faulty datetime values. Defaults to*) –
- **threshold** (*float*) – The minimum percentage of windows that need to have been able to infer a frequency. Lower values make this component more

- **0.8.** (*sensitive to recognizing numerous faulty datetime values. Defaults to*) –
- **random_seed** (*int*) – Seed for the random number generator. This transformer performs the same regardless of the random seed provided.
- **0.** (*Defaults to*) –

**Raises** **ValueError** – if the frequency_payload parameter has not been passed a tuple

**Attributes**

| | |
|---|---|
| **hyper-parameter_ranges** | {} |
| **modifies_features** | True |
| **modifies_target** | True |
| **name** | Time Series Regularizer |
| **training_only** | True |

**Methods**

| | |
|---|---|
| *clone* | Constructs a new component with the same parameters and random state. |
| *default_parameters* | Returns the default parameters for this component. |
| *describe* | Describe a component and its parameters. |
| *fit* | Fits the TimeSeriesRegularizer. |
| *fit_transform* | Fits on X and transforms X. |
| *load* | Loads component at file path. |
| *needs_fitting* | Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances. |
| *parameters* | Returns the parameters which were used to initialize the component. |
| *save* | Saves component at file path. |
| *transform* | Regularizes a dataframe and target data to an inferrable offset frequency. |
| *update_parameters* | Updates the parameter dictionary of the component. |

**clone**(*self*)

Constructs a new component with the same parameters and random state.

**Returns** A new instance of this component with identical parameters and random state.

**default_parameters**(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

**Returns** Default parameters for this component.

**Return type** dict

**describe**(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

**Parameters**

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description as dictionary in the format {"name": name, "parameters": parameters}

**Returns** Returns dictionary if return_dict is True, else None.

**Return type** None or dict

**fit**(*self*, *X*, *y=None*)

Fits the TimeSeriesRegularizer.

**Parameters**

- **X** (*pd.DataFrame*) – The input training data of shape [n_samples, n_features].
- **y** (*pd.Series, optional*) – The target training data of length [n_samples].

**Returns** self

**Raises**

- **ValueError** – if self.time_index is None, if X and y have different lengths, if *time_index* in X does not have an offset frequency that can be estimated
- **TypeError** – if the *time_index* column is not of type Datetime
- **KeyError** – if the *time_index* column doesn't exist

**fit_transform**(*self*, *X*, *y=None*)

Fits on X and transforms X.

**Parameters**

- **X** (*pd.DataFrame*) – Data to fit and transform.
- **y** (*pd.Series*) – Target data.

**Returns** Transformed X.

**Return type** pd.DataFrame

**Raises** **MethodPropertyNotFoundError** – If transformer does not have a transform method or a component_obj that implements transform.

**static load**(*file_path*)

Loads component at file path.

**Parameters** **file_path** (*str*) – Location to load file.

**Returns** ComponentBase object

**needs_fitting**(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need to be fit or whose fit methods do nothing.

**Returns** True.

**property parameters**(*self*)

Returns the parameters which were used to initialize the component.

**save**(*self*, *file_path*, *pickle_protocol=cloudpickle.DEFAULT_PROTOCOL*)

Saves component at file path.

**Parameters**

- **file_path** (*str*) – Location to save file.
- **pickle_protocol** (*int*) – The pickle data stream format.

**transform**(*self*, *X*, *y=None*)

Regularizes a dataframe and target data to an inferrable offset frequency.

A 'clean' X and y (if y was passed in) are created based on an inferrable offset frequency and matching datetime values with the original X and y are imputed into the clean X and y. Datetime values identified as misaligned are shifted into their appropriate position.

**Parameters**

- **X** (*pd.DataFrame*) – The input training data of shape [n_samples, n_features].
- **y** (*pd.Series, optional*) – The target training data of length [n_samples].

**Returns** Data with an inferrable *time_index* offset frequency.

**Return type** (pd.DataFrame, pd.Series)

**update_parameters**(*self*, *update_dict*, *reset_fit=True*)

Updates the parameter dictionary of the component.

**Parameters**

- **update_dict** (*dict*) – A dict of parameters to update.
- **reset_fit** (*bool, optional*) – If True, will set *_is_fitted* to False.

## transform_primitive_components

Components that extract features from the input data.

### Module Contents

### Classes Summary

| | |
|---|---|
| *EmailFeaturizer* | Transformer that can automatically extract features from emails. |
| *URLFeaturizer* | Transformer that can automatically extract features from URL. |

## Contents

**class** evalml.pipelines.components.transformers.preprocessing.transform_primitive_components.**EmailFeatur**

Transformer that can automatically extract features from emails.

**Parameters** **random_seed** (*int*) – Seed for the random number generator. Defaults to 0.

**Attributes**

| | |
|---|---|
| **hyper-parameter_ranges** | {} |
| **modifies_features** | True |
| **modifies_target** | False |
| **name** | Email Featurizer |
| **training_only** | False |

**Methods**

| | |
|---|---|
| *clone* | Constructs a new component with the same parameters and random state. |
| *default_parameters* | Returns the default parameters for this component. |
| *describe* | Describe a component and its parameters. |
| *fit* | Fits component to data. |
| *fit_transform* | Fits on X and transforms X. |
| *load* | Loads component at file path. |
| *needs_fitting* | Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances. |
| *parameters* | Returns the parameters which were used to initialize the component. |
| *save* | Saves component at file path. |
| *transform* | Transforms data X. |
| *update_parameters* | Updates the parameter dictionary of the component. |

**clone**(*self*)

Constructs a new component with the same parameters and random state.

**Returns** A new instance of this component with identical parameters and random state.

**default_parameters**(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

**Returns** Default parameters for this component.

**Return type** dict

**describe**(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

**Parameters**

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description as dictionary in the format {"name": name, "parameters": parameters}

**Returns** Returns dictionary if return_dict is True, else None.

**Return type** None or dict

**fit**(*self*, *X*, *y=None*)

Fits component to data.

**Parameters**

- **X** (*pd.DataFrame*) – The input training data of shape [n_samples, n_features]
- **y** (*pd.Series, optional*) – The target training data of length [n_samples]

**Returns** self

**Raises** **MethodPropertyNotFoundError** – If component does not have a fit method or a component_obj that implements fit.

**fit_transform**(*self*, *X*, *y=None*)

Fits on X and transforms X.

**Parameters**

- **X** (*pd.DataFrame*) – Data to fit and transform.
- **y** (*pd.Series*) – Target data.

**Returns** Transformed X.

**Return type** pd.DataFrame

**Raises** **MethodPropertyNotFoundError** – If transformer does not have a transform method or a component_obj that implements transform.

**static load**(*file_path*)

Loads component at file path.

**Parameters** **file_path** (*str*) – Location to load file.

**Returns** ComponentBase object

**needs_fitting**(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need to be fit or whose fit methods do nothing.

**Returns** True.

**property parameters**(*self*)

Returns the parameters which were used to initialize the component.

**save**(*self*, *file_path*, *pickle_protocol=cloudpickle.DEFAULT_PROTOCOL*)

Saves component at file path.

**Parameters**

- **file_path** (*str*) – Location to save file.
- **pickle_protocol** (*int*) – The pickle data stream format.

**transform**(*self*, *X*, *y=None*)

Transforms data X.

**Parameters**

- **X** (*pd.DataFrame*) – Data to transform.
- **y** (*pd.Series, optional*) – Target data.

**Returns** Transformed X

**Return type** pd.DataFrame

**Raises** **MethodPropertyNotFoundError** – If transformer does not have a transform method or a component_obj that implements transform.

**update_parameters**(*self*, *update_dict*, *reset_fit=True*)

Updates the parameter dictionary of the component.

**Parameters**

- **update_dict** (*dict*) – A dict of parameters to update.
- **reset_fit** (*bool, optional*) – If True, will set *_is_fitted* to False.

**class** evalml.pipelines.components.transformers.preprocessing.transform_primitive_components.**URLFeaturiz**

Transformer that can automatically extract features from URL.

**Parameters** **random_seed** (*int*) – Seed for the random number generator. Defaults to 0.

**Attributes**

| | |
|---|---|
| **hyper-parameter_ranges** | {} |
| **modifies_features** | True |
| **modifies_target** | False |
| **name** | URL Featurizer |
| **training_only** | False |

**Methods**

| | |
|---|---|
| *clone* | Constructs a new component with the same parameters and random state. |
| *default_parameters* | Returns the default parameters for this component. |
| *describe* | Describe a component and its parameters. |
| *fit* | Fits component to data. |
| *fit_transform* | Fits on X and transforms X. |
| *load* | Loads component at file path. |
| *needs_fitting* | Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances. |
| *parameters* | Returns the parameters which were used to initialize the component. |
| *save* | Saves component at file path. |
| *transform* | Transforms data X. |
| *update_parameters* | Updates the parameter dictionary of the component. |

**clone**(*self*)

Constructs a new component with the same parameters and random state.

**Returns** A new instance of this component with identical parameters and random state.

**default_parameters**(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

**Returns** Default parameters for this component.

**Return type** dict

**describe**(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

**Parameters**

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description as dictionary in the format {"name": name, "parameters": parameters}

**Returns** Returns dictionary if return_dict is True, else None.

**Return type** None or dict

**fit**(*self*, *X*, *y=None*)

Fits component to data.

**Parameters**

- **X** (*pd.DataFrame*) – The input training data of shape [n_samples, n_features]
- **y** (*pd.Series, optional*) – The target training data of length [n_samples]

**Returns** self

**Raises** **MethodPropertyNotFoundError** – If component does not have a fit method or a component_obj that implements fit.

**fit_transform**(*self*, *X*, *y=None*)

Fits on X and transforms X.

**Parameters**

- **X** (*pd.DataFrame*) – Data to fit and transform.
- **y** (*pd.Series*) – Target data.

**Returns** Transformed X.

**Return type** pd.DataFrame

**Raises** **MethodPropertyNotFoundError** – If transformer does not have a transform method or a component_obj that implements transform.

**static load**(*file_path*)

Loads component at file path.

**Parameters** **file_path** (*str*) – Location to load file.

**Returns** ComponentBase object

**needs_fitting**(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need to be fit or whose fit methods do nothing.

**Returns** True.

**property parameters**(*self*)

Returns the parameters which were used to initialize the component.

**save**(*self*, *file_path*, *pickle_protocol=cloudpickle.DEFAULT_PROTOCOL*)

Saves component at file path.

**Parameters**

- **file_path** (*str*) – Location to save file.
- **pickle_protocol** (*int*) – The pickle data stream format.

**transform**(*self*, *X*, *y=None*)

Transforms data X.

**Parameters**

- **X** (*pd.DataFrame*) – Data to transform.
- **y** (*pd.Series, optional*) – Target data.

**Returns** Transformed X

**Return type** pd.DataFrame

**Raises** **MethodPropertyNotFoundError** – If transformer does not have a transform method or a component_obj that implements transform.

**update_parameters**(*self*, *update_dict*, *reset_fit=True*)

Updates the parameter dictionary of the component.

**Parameters**

- **update_dict** (*dict*) – A dict of parameters to update.
- **reset_fit** (*bool, optional*) – If True, will set *_is_fitted* to False.

### Package Contents

### Classes Summary

| | |
|---|---|
| *DateTimeFeaturizer* | Transformer that can automatically extract features from datetime columns. |
| *Decomposer* | Component that removes trends and seasonality from time series and returns the decomposed components. |
| *DFSTransformer* | Featuretools DFS component that generates features for the input features. |
| *DropNaNRowsTransformer* | Transformer to drop rows with NaN values. |
| *DropNullColumns* | Transformer to drop features whose percentage of NaN values exceeds a specified threshold. |
| *DropRowsTransformer* | Transformer to drop rows specified by row indices. |
| *EmailFeaturizer* | Transformer that can automatically extract features from emails. |
| *LogTransformer* | Applies a log transformation to the target data. |
| *LSA* | Transformer to calculate the Latent Semantic Analysis Values of text input. |
| *NaturalLanguageFeaturizer* | Transformer that can automatically featurize text columns using featuretools' nlp_primitives. |
| *PolynomialDecomposer* | Removes trends and seasonality from time series by fitting a polynomial and moving average to the data. |
| *ReplaceNullableTypes* | Transformer to replace features with the new nullable dtypes with a dtype that is compatible in EvalML. |
| *STLDecomposer* | Removes trends and seasonality from time series using the STL algorithm. |
| *TextTransformer* | Base class for all transformers working with text features. |
| *TimeSeriesFeaturizer* | Transformer that delays input features and target variable for time series problems. |
| *TimeSeriesRegularizer* | Transformer that regularizes an inconsistently spaced datetime column. |
| *URLFeaturizer* | Transformer that can automatically extract features from URL. |

### Contents

**class** evalml.pipelines.components.transformers.preprocessing.**DateTimeFeaturizer**(*features_to_extract=None*, *encode_as_categories=False*, *time_index=None*, *random_seed=0*, ***kwargs*)

Transformer that can automatically extract features from datetime columns.

**Parameters**

- **features_to_extract** (*list*) – List of features to extract. Valid options include "year", "month", "day_of_week", "hour". Defaults to None.

- **encode_as_categories** (*bool*) – Whether day-of-week and month features should be encoded as pandas "category" dtype. This allows OneHotEncoders to encode these features. Defaults to False.
- **time_index** (*str*) – Name of the column containing the datetime information used to order the data. Ignored.
- **random_seed** (*int*) – Seed for the random number generator. Defaults to 0.

**Attributes**

| | |
|---|---|
| **hyper-parameter_ranges** | {} |
| **modifies_features** | True |
| **modifies_target** | False |
| **name** | DateTime Featurizer |
| **training_only** | False |

**Methods**

| | |
|---|---|
| *clone* | Constructs a new component with the same parameters and random state. |
| *default_parameters* | Returns the default parameters for this component. |
| *describe* | Describe a component and its parameters. |
| *fit* | Fit the datetime featurizer component. |
| *fit_transform* | Fits on X and transforms X. |
| *get_feature_names* | Gets the categories of each datetime feature. |
| *load* | Loads component at file path. |
| *needs_fitting* | Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances. |
| *parameters* | Returns the parameters which were used to initialize the component. |
| *save* | Saves component at file path. |
| *transform* | Transforms data X by creating new features using existing DateTime columns, and then dropping those DateTime columns. |
| *update_parameters* | Updates the parameter dictionary of the component. |

**clone**(*self*)

Constructs a new component with the same parameters and random state.

**Returns** A new instance of this component with identical parameters and random state.

**default_parameters**(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

**Returns** Default parameters for this component.

**Return type** dict

**describe**(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

**Parameters**

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description as dictionary in the format {"name": name, "parameters": parameters}

**Returns** Returns dictionary if return_dict is True, else None.

**Return type** None or dict

**fit**(*self*, *X*, *y=None*)

Fit the datetime featurizer component.

**Parameters**

- **X** (*pd.DataFrame*) – Input features.
- **y** (*pd.Series, optional*) – Target data. Ignored.

**Returns** self

**fit_transform**(*self*, *X*, *y=None*)

Fits on X and transforms X.

**Parameters**

- **X** (*pd.DataFrame*) – Data to fit and transform.
- **y** (*pd.Series*) – Target data.

**Returns** Transformed X.

**Return type** pd.DataFrame

**Raises** **MethodPropertyNotFoundError** – If transformer does not have a transform method or a component_obj that implements transform.

**get_feature_names**(*self*)

Gets the categories of each datetime feature.

**Returns**

**Dictionary, where each key-value pair is a column name and a dictionary** mapping the unique feature values to their integer encoding.

**Return type** dict

**static load**(*file_path*)

Loads component at file path.

**Parameters** **file_path** (*str*) – Location to load file.

**Returns** ComponentBase object

**needs_fitting**(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need to be fit or whose fit methods do nothing.

**Returns** True.

**property parameters**(*self*)

Returns the parameters which were used to initialize the component.

**save**(*self*, *file_path*, *pickle_protocol=cloudpickle.DEFAULT_PROTOCOL*)

Saves component at file path.

**Parameters**

- **file_path** (*str*) – Location to save file.
- **pickle_protocol** (*int*) – The pickle data stream format.

**transform**(*self*, *X*, *y=None*)

Transforms data X by creating new features using existing DateTime columns, and then dropping those DateTime columns.

**Parameters**

- **X** (*pd.DataFrame*) – Input features.
- **y** (*pd.Series, optional*) – Ignored.

**Returns** Transformed X

**Return type** pd.DataFrame

**update_parameters**(*self*, *update_dict*, *reset_fit=True*)

Updates the parameter dictionary of the component.

**Parameters**

- **update_dict** (*dict*) – A dict of parameters to update.
- **reset_fit** (*bool, optional*) – If True, will set *_is_fitted* to False.

**class** evalml.pipelines.components.transformers.preprocessing.**Decomposer**(*component_obj=None*, *random_seed: int = 0*, *degree: int = 1*, *period: int = - 1*, *seasonal_smoother: int = 7*, *time_index: str = None*, ***kwargs*)

Component that removes trends and seasonality from time series and returns the decomposed components.

**Parameters**

- **parameters** (*dict*) – Dictionary of parameters to pass to component object.
- **component_obj** (*class*) – Instance of a detrender/deseasonalizer class.
- **random_seed** (*int*) – Seed for the random number generator. Defaults to 0.
- **degree** (*int*) – Currently the degree of the PolynomialDecomposer, not used for STLDecomposer.
- **period** (*int*) – The best guess, in units, for the period of the seasonal signal.
- **seasonal_smoother** (*int*) – The seasonal smoothing parameter for STLDecomposer, not used for PolynomialDecomposer.
- **time_index** (*str*) – The column name of the feature matrix (X) that the datetime information should be pulled from.

**Attributes**

| | |
|---|---|
| **hyper-parameter_ranges** | None |
| **invalid_frequencies** | [] |
| **modifies_features** | False |
| **modifies_target** | True |
| **name** | Decomposer |
| **needs_fitting** | True |
| **training_only** | False |

**Methods**

| | |
|---|---|
| *clone* | Constructs a new component with the same parameters and random state. |
| *default_parameters* | Returns the default parameters for this component. |
| *describe* | Describe a component and its parameters. |
| *determine_periodicity* | Function that uses autocorrelative methods to determine the likely most signficant period of the seasonal signal. |
| *fit* | Fits component to data. |
| *fit_transform* | Removes fitted trend and seasonality from target variable. |
| *get_trend_dataframe* | Return a list of dataframes, each with 3 columns: trend, seasonality, residual. |
| *inverse_transform* | Add the trend + seasonality back to y. |
| *is_freq_valid* | Determines if the given string represents a valid frequency for this decomposer. |
| *load* | Loads component at file path. |
| *parameters* | Returns the parameters which were used to initialize the component. |
| *plot_decomposition* | Plots the decomposition of the target signal. |
| *save* | Saves component at file path. |
| *set_period* | Function to set the component's seasonal period based on the target's seasonality. |
| *transform* | Transforms data X. |
| *update_parameters* | Updates the parameter dictionary of the component. |

**clone**(*self*)

Constructs a new component with the same parameters and random state.

**Returns** A new instance of this component with identical parameters and random state.

**default_parameters**(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

**Returns** Default parameters for this component.

**Return type** dict

**describe**(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

**Parameters**

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description as dictionary in the format {"name": name, "parameters": parameters}

**Returns** Returns dictionary if return_dict is True, else None.

**Return type** None or dict

**classmethod determine_periodicity**(*cls*, *X: pandas.DataFrame*, *y: pandas.Series*, *acf_threshold: float = 0.01*, *rel_max_order: int = 5*)

Function that uses autocorrelative methods to determine the likely most signficant period of the seasonal signal.

**Parameters**

- **X** (*pandas.DataFrame*) – The feature data of the time series problem.
- **y** (*pandas.Series*) – The target data of a time series problem.
- **acf_threshold** (*float*) – The threshold for the autocorrelation function to determine the period. Any values below the threshold are considered to be 0 and will not be considered for the period. Defaults to 0.01.
- **rel_max_order** (*int*) – The order of the relative maximum to determine the period. Defaults to 5.

**Returns**

**The integer number of entries in time series data over which the seasonal part of the target data** repeats. If the time series data is in days, then this is the number of days that it takes the target's seasonal signal to repeat. Note: the target data can contain multiple seasonal signals. This function will only return the stronger. E.g. if the target has both weekly and yearly seasonality, the function may return either "7" or "365", depending on which seasonality is more strongly autocorrelated. If no period is detected, returns None.

**Return type** int

**fit**(*self*, *X*, *y=None*)

Fits component to data.

**Parameters**

- **X** (*pd.DataFrame*) – The input training data of shape [n_samples, n_features]
- **y** (*pd.Series, optional*) – The target training data of length [n_samples]

**Returns** self

**Raises** **MethodPropertyNotFoundError** – If component does not have a fit method or a component_obj that implements fit.

**fit_transform**(*self*, *X: pandas.DataFrame*, *y: pandas.Series = None*) → tuple[pandas.DataFrame, pandas.Series]

Removes fitted trend and seasonality from target variable.

**Parameters**

- **X** (*pd.DataFrame, optional*) – Ignored.
- **y** (*pd.Series*) – Target variable to detrend and deseasonalize.

**Returns**

**The first element are the input features returned without modification.** The second element is the target variable y with the fitted trend removed.

**Return type** tuple of pd.DataFrame, pd.Series

**abstract get_trend_dataframe**(*self*, *y: pandas.Series*)

Return a list of dataframes, each with 3 columns: trend, seasonality, residual.

**abstract inverse_transform**(*self*, *y: pandas.Series*)

Add the trend + seasonality back to y.

**classmethod is_freq_valid**(*cls*, *freq: str*)

Determines if the given string represents a valid frequency for this decomposer.

**Parameters** **freq** (*str*) – A frequency to validate. See the pandas docs at https://pandas.pydata.org/pandas-docs/stable/user_guide/timeseries.html#offset-aliases for options.

**Returns** boolean representing whether the frequency is valid or not.

**static load**(*file_path*)

Loads component at file path.

**Parameters** **file_path** (*str*) – Location to load file.

**Returns** ComponentBase object

**property parameters**(*self*)

Returns the parameters which were used to initialize the component.

**plot_decomposition**(*self*, *X: pandas.DataFrame*, *y: pandas.Series*, *show: bool = False*) → tuple[matplotlib.pyplot.Figure, list]

Plots the decomposition of the target signal.

**Parameters**

- **X** (*pd.DataFrame*) – Input data with time series data in index.
- **y** (*pd.Series or pd.DataFrame*) – Target variable data provided as a Series for univariate problems or a DataFrame for multivariate problems.
- **show** (*bool*) – Whether to display the plot or not. Defaults to False.

**Returns**

**The figure and axes that have the decompositions** plotted on them

**Return type** matplotlib.pyplot.Figure, list[matplotlib.pyplot.Axes]

**save**(*self*, *file_path*, *pickle_protocol=cloudpickle.DEFAULT_PROTOCOL*)

Saves component at file path.

**Parameters**

- **file_path** (*str*) – Location to save file.
- **pickle_protocol** (*int*) – The pickle data stream format.

**set_period**(*self*, *X: pandas.DataFrame*, *y: pandas.Series*, *acf_threshold: float = 0.01*, *rel_max_order: int = 5*)

Function to set the component's seasonal period based on the target's seasonality.

**Parameters**

- **X** (*pandas.DataFrame*) – The feature data of the time series problem.
- **y** (*pandas.Series*) – The target data of a time series problem.
- **acf_threshold** (*float*) – The threshold for the autocorrelation function to determine the period. Any values below the threshold are considered to be 0 and will not be considered for the period. Defaults to 0.01.
- **rel_max_order** (*int*) – The order of the relative maximum to determine the period. Defaults to 5.

**abstract transform**(*self*, *X*, *y=None*)

Transforms data X.

**Parameters**

- **X** (*pd.DataFrame*) – Data to transform.
- **y** (*pd.Series, optional*) – Target data.

**Returns** Transformed X

**Return type** pd.DataFrame

**Raises** **MethodPropertyNotFoundError** – If transformer does not have a transform method or a component_obj that implements transform.

**update_parameters**(*self*, *update_dict*, *reset_fit=True*)

Updates the parameter dictionary of the component.

**Parameters**

- **update_dict** (*dict*) – A dict of parameters to update.
- **reset_fit** (*bool, optional*) – If True, will set *_is_fitted* to False.

**class** evalml.pipelines.components.transformers.preprocessing.**DFSTransformer**(*index='index'*, *features=None*, *random_seed=0*, ***kwargs*)

Featuretools DFS component that generates features for the input features.

**Parameters**

- **index** (*string*) – The name of the column that contains the indices. If no column with this name exists, then featuretools.EntitySet() creates a column with this name to serve as the index column. Defaults to 'index'.
- **random_seed** (*int*) – Seed for the random number generator. Defaults to 0.
- **features** (*list*) – List of features to run DFS on. Defaults to None. Features will only be computed if the columns used by the feature exist in the input and if the feature itself is not in input. If features is an empty list, no transformation will occur to inputted data.

**Attributes**

| | |
|---|---|
| **hyper-parameter_ranges** | {} |
| **modifies_features** | True |
| **modifies_target** | False |
| **name** | DFS Transformer |
| **training_only** | False |

**Methods**

| | |
|---|---|
| *clone* | Constructs a new component with the same parameters and random state. |
| *contains_pre_existing_features* | Determines whether or not features from a DFS Transformer match pipeline input features. |
| *default_parameters* | Returns the default parameters for this component. |
| *describe* | Describe a component and its parameters. |
| *fit* | Fits the DFSTransformer Transformer component. |
| *fit_transform* | Fits on X and transforms X. |
| *load* | Loads component at file path. |
| *needs_fitting* | Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances. |
| *parameters* | Returns the parameters which were used to initialize the component. |
| *save* | Saves component at file path. |
| *transform* | Computes the feature matrix for the input X using featuretools' dfs algorithm. |
| *update_parameters* | Updates the parameter dictionary of the component. |

**clone**(*self*)

Constructs a new component with the same parameters and random state.

**Returns** A new instance of this component with identical parameters and random state.

**static contains_pre_existing_features**(*dfs_features: Optional[List[featuretools.feature_base.FeatureBase]], input_feature_names: List[str], target: Optional[str] = None*)

Determines whether or not features from a DFS Transformer match pipeline input features.

**Parameters**

- **dfs_features** (*Optional[List[FeatureBase]]*) – List of features output from a DFS Transformer.
- **input_feature_names** (*List[str]*) – List of input features into the DFS Transformer.
- **target** (*Optional[str]*) – The target whose values we are trying to predict. This is used to know which column to ignore if the target column is present in the list of features in the DFS Transformer's parameters.

**default_parameters**(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

**Returns** Default parameters for this component.

**Return type** dict

**describe**(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

**Parameters**

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description as dictionary in the format {"name": name, "parameters": parameters}

**Returns** Returns dictionary if return_dict is True, else None.

**Return type** None or dict

**fit**(*self*, *X*, *y=None*)

Fits the DFSTransformer Transformer component.

**Parameters**

- **X** (*pd.DataFrame, np.array*) – The input data to transform, of shape [n_samples, n_features].
- **y** (*pd.Series*) – The target training data of length [n_samples].

**Returns** self

**fit_transform**(*self*, *X*, *y=None*)

Fits on X and transforms X.

**Parameters**

- **X** (*pd.DataFrame*) – Data to fit and transform.
- **y** (*pd.Series*) – Target data.

**Returns** Transformed X.

**Return type** pd.DataFrame

**Raises** **MethodPropertyNotFoundError** – If transformer does not have a transform method or a component_obj that implements transform.

**static load**(*file_path*)

Loads component at file path.

**Parameters** **file_path** (*str*) – Location to load file.

**Returns** ComponentBase object

**needs_fitting**(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need to be fit or whose fit methods do nothing.

**Returns** True.

**property parameters**(*self*)

Returns the parameters which were used to initialize the component.

**save**(*self*, *file_path*, *pickle_protocol=cloudpickle.DEFAULT_PROTOCOL*)

Saves component at file path.

**Parameters**

- **file_path** (*str*) – Location to save file.
- **pickle_protocol** (*int*) – The pickle data stream format.

**transform**(*self*, *X*, *y=None*)

Computes the feature matrix for the input X using featuretools' dfs algorithm.

**Parameters**

- **X** (*pd.DataFrame or np.ndarray*) – The input training data to transform. Has shape [n_samples, n_features]
- **y** (*pd.Series, optional*) – Ignored.

**Returns** Feature matrix

**Return type** pd.DataFrame

**update_parameters**(*self*, *update_dict*, *reset_fit=True*)

Updates the parameter dictionary of the component.

**Parameters**

- **update_dict** (*dict*) – A dict of parameters to update.
- **reset_fit** (*bool, optional*) – If True, will set *_is_fitted* to False.

**class** evalml.pipelines.components.transformers.preprocessing.**DropNaNRowsTransformer**(*parameters=None*, *component_obj=None*, *random_seed=0*, ***kwargs*)

Transformer to drop rows with NaN values.

**Parameters** **random_seed** (*int*) – Seed for the random number generator. Is not used by this component. Defaults to 0.

**Attributes**

| | |
|---|---|
| **hyperparameter_ranges** | {} |
| **modifies_features** | True |
| **modifies_target** | True |
| **name** | Drop NaN Rows Transformer |
| **training_only** | False |

**Methods**

| | |
|---|---|
| *clone* | Constructs a new component with the same parameters and random state. |
| *default_parameters* | Returns the default parameters for this component. |
| *describe* | Describe a component and its parameters. |
| *fit* | Fits component to data. |
| *fit_transform* | Fits on X and transforms X. |
| *load* | Loads component at file path. |
| *needs_fitting* | Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances. |
| *parameters* | Returns the parameters which were used to initialize the component. |
| *save* | Saves component at file path. |
| *transform* | Transforms data using fitted component. |
| *update_parameters* | Updates the parameter dictionary of the component. |

**clone**(*self*)

Constructs a new component with the same parameters and random state.

**Returns** A new instance of this component with identical parameters and random state.

**default_parameters**(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

**Returns** Default parameters for this component.

**Return type** dict

**describe**(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

**Parameters**

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description as dictionary in the format {"name": name, "parameters": parameters}

**Returns** Returns dictionary if return_dict is True, else None.

**Return type** None or dict

**fit**(*self*, *X*, *y=None*)

Fits component to data.

**Parameters**

- **X** (*pd.DataFrame*) – The input training data of shape [n_samples, n_features].
- **y** (*pd.Series, optional*) – The target training data of length [n_samples].

**Returns** self

**fit_transform**(*self*, *X*, *y=None*)

Fits on X and transforms X.

**Parameters**

- **X** (*pd.DataFrame*) – Data to fit and transform.

- **y** (*pd.Series*) – Target data.

**Returns** Transformed X.

**Return type** pd.DataFrame

**Raises** **MethodPropertyNotFoundError** – If transformer does not have a transform method or a component_obj that implements transform.

**static load**(*file_path*)

Loads component at file path.

**Parameters** **file_path** (*str*) – Location to load file.

**Returns** ComponentBase object

**needs_fitting**(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need to be fit or whose fit methods do nothing.

**Returns** True.

**property parameters**(*self*)

Returns the parameters which were used to initialize the component.

**save**(*self*, *file_path*, *pickle_protocol=cloudpickle.DEFAULT_PROTOCOL*)

Saves component at file path.

**Parameters**

- **file_path** (*str*) – Location to save file.
- **pickle_protocol** (*int*) – The pickle data stream format.

**transform**(*self*, *X*, *y=None*)

Transforms data using fitted component.

**Parameters**

- **X** (*pd.DataFrame*) – Features.
- **y** (*pd.Series, optional*) – Target data.

**Returns** Data with NaN rows dropped.

**Return type** (pd.DataFrame, pd.Series)

**update_parameters**(*self*, *update_dict*, *reset_fit=True*)

Updates the parameter dictionary of the component.

**Parameters**

- **update_dict** (*dict*) – A dict of parameters to update.
- **reset_fit** (*bool, optional*) – If True, will set *_is_fitted* to False.

**class** evalml.pipelines.components.transformers.preprocessing.**DropNullColumns**(*pct_null_threshold=1.0*, *random_seed=0*, ***kwargs*)

Transformer to drop features whose percentage of NaN values exceeds a specified threshold.

**Parameters**

- **pct_null_threshold** (*float*) – The percentage of NaN values in an input feature to drop. Must be a value between [0, 1] inclusive. If equal to 0.0, will drop columns with any null values. If equal to 1.0, will drop columns with all null values. Defaults to 0.95.
- **random_seed** (*int*) – Seed for the random number generator. Defaults to 0.

**Attributes**

| | |
|---|---|
| **hyper-parameter_ranges** | {} |
| **modifies_features** | True |
| **modifies_target** | False |
| **name** | Drop Null Columns Transformer |
| **training_only** | False |

**Methods**

| | |
|---|---|
| *clone* | Constructs a new component with the same parameters and random state. |
| *default_parameters* | Returns the default parameters for this component. |
| *describe* | Describe a component and its parameters. |
| *fit* | Fits component to data. |
| *fit_transform* | Fits on X and transforms X. |
| *load* | Loads component at file path. |
| *needs_fitting* | Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances. |
| *parameters* | Returns the parameters which were used to initialize the component. |
| *save* | Saves component at file path. |
| *transform* | Transforms data X by dropping columns that exceed the threshold of null values. |
| *update_parameters* | Updates the parameter dictionary of the component. |

**clone**(*self*)

Constructs a new component with the same parameters and random state.

**Returns** A new instance of this component with identical parameters and random state.

**default_parameters**(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

**Returns** Default parameters for this component.

**Return type** dict

**describe**(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

**Parameters**

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description as dictionary in the format {"name": name, "parameters": parameters}

**Returns** Returns dictionary if return_dict is True, else None.

**Return type** None or dict

**fit**(*self*, *X*, *y=None*)

Fits component to data.

**Parameters**

- **X** (*pd.DataFrame*) – The input training data of shape [n_samples, n_features].
- **y** (*pd.Series, optional*) – The target training data of length [n_samples].

**Returns** self

**fit_transform**(*self*, *X*, *y=None*)

Fits on X and transforms X.

**Parameters**

- **X** (*pd.DataFrame*) – Data to fit and transform.
- **y** (*pd.Series*) – Target data.

**Returns** Transformed X.

**Return type** pd.DataFrame

**Raises** **MethodPropertyNotFoundError** – If transformer does not have a transform method or a component_obj that implements transform.

**static load**(*file_path*)

Loads component at file path.

**Parameters** **file_path** (*str*) – Location to load file.

**Returns** ComponentBase object

**needs_fitting**(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need to be fit or whose fit methods do nothing.

**Returns** True.

**property parameters**(*self*)

Returns the parameters which were used to initialize the component.

**save**(*self*, *file_path*, *pickle_protocol=cloudpickle.DEFAULT_PROTOCOL*)

Saves component at file path.

**Parameters**

- **file_path** (*str*) – Location to save file.
- **pickle_protocol** (*int*) – The pickle data stream format.

**transform**(*self*, *X*, *y=None*)

Transforms data X by dropping columns that exceed the threshold of null values.

**Parameters**

- **X** (*pd.DataFrame*) – Data to transform
- **y** (*pd.Series, optional*) – Ignored.

**Returns** Transformed X

**Return type** pd.DataFrame

**update_parameters**(*self*, *update_dict*, *reset_fit=True*)

Updates the parameter dictionary of the component.

**Parameters**

- **update_dict** (*dict*) – A dict of parameters to update.
- **reset_fit** (*bool, optional*) – If True, will set *_is_fitted* to False.

**class** evalml.pipelines.components.transformers.preprocessing.**DropRowsTransformer**(*indices_to_drop=None*, *random_seed=0*)

Transformer to drop rows specified by row indices.

**Parameters**

- **indices_to_drop** (*list*) – List of indices to drop in the input data. Defaults to None.
- **random_seed** (*int*) – Seed for the random number generator. Is not used by this component. Defaults to 0.

**Attributes**

| | |
|---|---|
| **hyperparameter_ranges** | {} |
| **modifies_features** | True |
| **modifies_target** | True |
| **name** | Drop Rows Transformer |
| **training_only** | True |

**Methods**

| | |
|---|---|
| *clone* | Constructs a new component with the same parameters and random state. |
| *default_parameters* | Returns the default parameters for this component. |
| *describe* | Describe a component and its parameters. |
| *fit* | Fits component to data. |
| *fit_transform* | Fits on X and transforms X. |
| *load* | Loads component at file path. |
| *needs_fitting* | Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances. |
| *parameters* | Returns the parameters which were used to initialize the component. |
| *save* | Saves component at file path. |
| *transform* | Transforms data using fitted component. |
| *update_parameters* | Updates the parameter dictionary of the component. |

**clone**(*self*)

Constructs a new component with the same parameters and random state.

**Returns** A new instance of this component with identical parameters and random state.

**default_parameters**(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

**Returns** Default parameters for this component.

**Return type** dict

**describe**(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

**Parameters**

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description as dictionary in the format {"name": name, "parameters": parameters}

**Returns** Returns dictionary if return_dict is True, else None.

**Return type** None or dict

**fit**(*self*, *X*, *y=None*)

Fits component to data.

**Parameters**

- **X** (*pd.DataFrame*) – The input training data of shape [n_samples, n_features].
- **y** (*pd.Series, optional*) – The target training data of length [n_samples].

**Returns** self

**Raises** **ValueError** – If indices to drop do not exist in input features or target.

**fit_transform**(*self*, *X*, *y=None*)

Fits on X and transforms X.

**Parameters**

- **X** (*pd.DataFrame*) – Data to fit and transform.
- **y** (*pd.Series*) – Target data.

**Returns** Transformed X.

**Return type** pd.DataFrame

**Raises** **MethodPropertyNotFoundError** – If transformer does not have a transform method or a component_obj that implements transform.

**static load**(*file_path*)

Loads component at file path.

**Parameters** **file_path** (*str*) – Location to load file.

**Returns** ComponentBase object

**needs_fitting**(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need to be fit or whose fit methods do nothing.

**Returns** True.

**property parameters**(*self*)

Returns the parameters which were used to initialize the component.

**save**(*self*, *file_path*, *pickle_protocol=cloudpickle.DEFAULT_PROTOCOL*)

Saves component at file path.

**Parameters**

- **file_path** (*str*) – Location to save file.
- **pickle_protocol** (*int*) – The pickle data stream format.

**transform**(*self*, *X*, *y=None*)

Transforms data using fitted component.

**Parameters**

- **X** (*pd.DataFrame*) – Features.
- **y** (*pd.Series, optional*) – Target data.

**Returns** Data with row indices dropped.

**Return type** (pd.DataFrame, pd.Series)

**update_parameters**(*self*, *update_dict*, *reset_fit=True*)

Updates the parameter dictionary of the component.

**Parameters**

- **update_dict** (*dict*) – A dict of parameters to update.
- **reset_fit** (*bool, optional*) – If True, will set *_is_fitted* to False.

**class** evalml.pipelines.components.transformers.preprocessing.**EmailFeaturizer**(*random_seed=0*, ***kwargs*)

Transformer that can automatically extract features from emails.

**Parameters** **random_seed** (*int*) – Seed for the random number generator. Defaults to 0.

**Attributes**

| | |
|---|---|
| **hyper-parameter_ranges** | {} |
| **modifies_features** | True |
| **modifies_target** | False |
| **name** | Email Featurizer |
| **training_only** | False |

**Methods**

| | |
|---|---|
| *clone* | Constructs a new component with the same parameters and random state. |
| *default_parameters* | Returns the default parameters for this component. |
| *describe* | Describe a component and its parameters. |
| *fit* | Fits component to data. |
| *fit_transform* | Fits on X and transforms X. |
| *load* | Loads component at file path. |
| *needs_fitting* | Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances. |
| *parameters* | Returns the parameters which were used to initialize the component. |
| *save* | Saves component at file path. |
| *transform* | Transforms data X. |
| *update_parameters* | Updates the parameter dictionary of the component. |

**clone**(*self*)

Constructs a new component with the same parameters and random state.

**Returns** A new instance of this component with identical parameters and random state.

**default_parameters**(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

**Returns** Default parameters for this component.

**Return type** dict

**describe**(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

**Parameters**

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description as dictionary in the format {"name": name, "parameters": parameters}

**Returns** Returns dictionary if return_dict is True, else None.

**Return type** None or dict

`fit`(*self*, *X*, *y=None*)

Fits component to data.

**Parameters**

- **X** (*pd.DataFrame*) – The input training data of shape [n_samples, n_features]
- **y** (*pd.Series, optional*) – The target training data of length [n_samples]

**Returns** self

**Raises** **MethodPropertyNotFoundError** – If component does not have a fit method or a component_obj that implements fit.

`fit_transform`(*self*, *X*, *y=None*)

Fits on X and transforms X.

**Parameters**

- **X** (*pd.DataFrame*) – Data to fit and transform.
- **y** (*pd.Series*) – Target data.

**Returns** Transformed X.

**Return type** pd.DataFrame

**Raises** **MethodPropertyNotFoundError** – If transformer does not have a transform method or a component_obj that implements transform.

`static load`(*file_path*)

Loads component at file path.

**Parameters** **file_path** (*str*) – Location to load file.

**Returns** ComponentBase object

`needs_fitting`(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need to be fit or whose fit methods do nothing.

**Returns** True.

`property parameters`(*self*)

Returns the parameters which were used to initialize the component.

`save`(*self*, *file_path*, *pickle_protocol=cloudpickle.DEFAULT_PROTOCOL*)

Saves component at file path.

**Parameters**

- **file_path** (*str*) – Location to save file.
- **pickle_protocol** (*int*) – The pickle data stream format.

`transform`(*self*, *X*, *y=None*)

Transforms data X.

**Parameters**

- **X** (*pd.DataFrame*) – Data to transform.

- **y** (*pd.Series, optional*) – Target data.

**Returns** Transformed X

**Return type** pd.DataFrame

**Raises** **MethodPropertyNotFoundError** – If transformer does not have a transform method or a component_obj that implements transform.

**update_parameters**(*self*, *update_dict*, *reset_fit=True*)

Updates the parameter dictionary of the component.

**Parameters**

- **update_dict** (*dict*) – A dict of parameters to update.
- **reset_fit** (*bool, optional*) – If True, will set *_is_fitted* to False.

**class** evalml.pipelines.components.transformers.preprocessing.**LogTransformer**(*random_seed=0*)

Applies a log transformation to the target data.

**Attributes**

| | |
|---|---|
| **hyper-parameter_ranges** | {} |
| **modifies_features** | False |
| **modifies_target** | True |
| **name** | Log Transformer |
| **training_only** | False |

**Methods**

| | |
|---|---|
| *clone* | Constructs a new component with the same parameters and random state. |
| *default_parameters* | Returns the default parameters for this component. |
| *describe* | Describe a component and its parameters. |
| *fit* | Fits the LogTransformer. |
| *fit_transform* | Log transforms the target variable. |
| *inverse_transform* | Apply exponential to target data. |
| *load* | Loads component at file path. |
| *needs_fitting* | Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances. |
| *parameters* | Returns the parameters which were used to initialize the component. |
| *save* | Saves component at file path. |
| *transform* | Log transforms the target variable. |
| *update_parameters* | Updates the parameter dictionary of the component. |

**clone**(*self*)

Constructs a new component with the same parameters and random state.

**Returns** A new instance of this component with identical parameters and random state.

**default_parameters**(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

**Returns** Default parameters for this component.

**Return type** dict

**describe**(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

**Parameters**

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description as dictionary in the format {"name": name, "parameters": parameters}

**Returns** Returns dictionary if return_dict is True, else None.

**Return type** None or dict

**fit**(*self*, *X*, *y=None*)

Fits the LogTransformer.

**Parameters**

- **X** (*pd.DataFrame or np.ndarray*) – Ignored.
- **y** (*pd.Series, optional*) – Ignored.

**Returns** self

**fit_transform**(*self*, *X*, *y=None*)

Log transforms the target variable.

**Parameters**

- **X** (*pd.DataFrame, optional*) – Ignored.
- **y** (*pd.Series*) – Target variable to log transform.

**Returns**

**The input features are returned without modification. The target** variable y is log transformed.

**Return type** tuple of pd.DataFrame, pd.Series

**inverse_transform**(*self*, *y*)

Apply exponential to target data.

**Parameters** **y** (*pd.Series*) – Target variable.

**Returns** Target with exponential applied.

**Return type** pd.Series

**static load**(*file_path*)

Loads component at file path.

**Parameters** **file_path** (*str*) – Location to load file.

**Returns** ComponentBase object

**needs_fitting**(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need to be fit or whose fit methods do nothing.

**Returns** True.

**property parameters**(*self*)

Returns the parameters which were used to initialize the component.

**save**(*self*, *file_path*, *pickle_protocol=cloudpickle.DEFAULT_PROTOCOL*)

Saves component at file path.

**Parameters**

- **file_path** (*str*) – Location to save file.
- **pickle_protocol** (*int*) – The pickle data stream format.

**transform**(*self*, *X*, *y=None*)

Log transforms the target variable.

**Parameters**

- **X** (*pd.DataFrame, optional*) – Ignored.
- **y** (*pd.Series*) – Target data to log transform.

**Returns**

**The input features are returned without modification. The target** variable y is log transformed.

**Return type** tuple of pd.DataFrame, pd.Series

**update_parameters**(*self*, *update_dict*, *reset_fit=True*)

Updates the parameter dictionary of the component.

**Parameters**

- **update_dict** (*dict*) – A dict of parameters to update.
- **reset_fit** (*bool, optional*) – If True, will set *_is_fitted* to False.

**class** evalml.pipelines.components.transformers.preprocessing.**LSA**(*random_seed=0*, ***kwargs*)

Transformer to calculate the Latent Semantic Analysis Values of text input.

**Parameters** **random_seed** (*int*) – Seed for the random number generator. Defaults to 0.

**Attributes**

| | |
|---|---|
| **hyper-parameter_ranges** | {} |
| **modifies_features** | True |
| **modifies_target** | False |
| **name** | LSA Transformer |
| **training_only** | False |

**Methods**

| | |
|---|---|
| *clone* | Constructs a new component with the same parameters and random state. |
| *default_parameters* | Returns the default parameters for this component. |
| *describe* | Describe a component and its parameters. |
| *fit* | Fits the input data. |
| *fit_transform* | Fits on X and transforms X. |
| *load* | Loads component at file path. |
| *needs_fitting* | Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances. |
| *parameters* | Returns the parameters which were used to initialize the component. |
| *save* | Saves component at file path. |
| *transform* | Transforms data X by applying the LSA pipeline. |
| *update_parameters* | Updates the parameter dictionary of the component. |

**clone**(*self*)

Constructs a new component with the same parameters and random state.

**Returns** A new instance of this component with identical parameters and random state.

**default_parameters**(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

**Returns** Default parameters for this component.

**Return type** dict

**describe**(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

**Parameters**

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description as dictionary in the format {"name": name, "parameters": parameters}

**Returns** Returns dictionary if return_dict is True, else None.

**Return type** None or dict

**fit**(*self*, *X*, *y=None*)

Fits the input data.

**Parameters**

- **X** (*pd.DataFrame*) – The data to transform.
- **y** (*pd.Series, optional*) – Ignored.

**Returns** self

**fit_transform**(*self*, *X*, *y=None*)

Fits on X and transforms X.

**Parameters**

- **X** (*pd.DataFrame*) – Data to fit and transform.
- **y** (*pd.Series*) – Target data.

**Returns** Transformed X.

**Return type** pd.DataFrame

**Raises** **MethodPropertyNotFoundError** – If transformer does not have a transform method or a component_obj that implements transform.

**static load**(*file_path*)

Loads component at file path.

**Parameters** **file_path** (*str*) – Location to load file.

**Returns** ComponentBase object

**needs_fitting**(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need to be fit or whose fit methods do nothing.

**Returns** True.

**property parameters**(*self*)

Returns the parameters which were used to initialize the component.

**save**(*self*, *file_path*, *pickle_protocol=cloudpickle.DEFAULT_PROTOCOL*)

Saves component at file path.

**Parameters**

- **file_path** (*str*) – Location to save file.
- **pickle_protocol** (*int*) – The pickle data stream format.

**transform**(*self*, *X*, *y=None*)

Transforms data X by applying the LSA pipeline.

**Parameters**

- **X** (*pd.DataFrame*) – The data to transform.
- **y** (*pd.Series, optional*) – Ignored.

**Returns**

**Transformed X. The original column is removed and replaced with two columns of the** format *LSA(original_column_name)[feature_number]*, where *feature_number* is 0 or 1.

**Return type** pd.DataFrame

**update_parameters**(*self*, *update_dict*, *reset_fit=True*)

Updates the parameter dictionary of the component.

**Parameters**

- **update_dict** (*dict*) – A dict of parameters to update.
- **reset_fit** (*bool, optional*) – If True, will set *_is_fitted* to False.

**class** evalml.pipelines.components.transformers.preprocessing.**NaturalLanguageFeaturizer**(*random_seed=0*, ***kwargs*)

Transformer that can automatically featurize text columns using featuretools' nlp_primitives.

Since models cannot handle non-numeric data, any text must be broken down into features that provide useful information about that text. This component splits each text column into several informative features: Diversity Score, Mean Characters per Word, Polarity Score, LSA (Latent Semantic Analysis), Number of Characters, and Number of Words. Calling transform on this component will replace any text columns in the given dataset with these numeric columns.

**Parameters** **random_seed** (*int*) – Seed for the random number generator. Defaults to 0.

**Attributes**

| | |
|---|---|
| **hyper-parameter_ranges** | {} |
| **modifies_features** | True |
| **modifies_target** | False |
| **name** | Natural Language Featurizer |
| **training_only** | False |

**Methods**

| | |
|---|---|
| *clone* | Constructs a new component with the same parameters and random state. |
| *default_parameters* | Returns the default parameters for this component. |
| *describe* | Describe a component and its parameters. |
| *fit* | Fits component to data. |
| *fit_transform* | Fits on X and transforms X. |
| *load* | Loads component at file path. |
| *needs_fitting* | Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances. |
| *parameters* | Returns the parameters which were used to initialize the component. |
| *save* | Saves component at file path. |
| *transform* | Transforms data X by creating new features using existing text columns. |
| *update_parameters* | Updates the parameter dictionary of the component. |

**clone**(*self*)

Constructs a new component with the same parameters and random state.

**Returns** A new instance of this component with identical parameters and random state.

**default_parameters**(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

**Returns** Default parameters for this component.

**Return type** dict

**describe**(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

**Parameters**

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description as dictionary in the format {"name": name, "parameters": parameters}

**Returns** Returns dictionary if return_dict is True, else None.

**Return type** None or dict

**fit**(*self*, *X*, *y=None*)

Fits component to data.

**Parameters**

- **X** (*pd.DataFrame or np.ndarray*) – The input training data of shape [n_samples, n_features]
- **y** (*pd.Series*) – The target training data of length [n_samples]

**Returns** self

**fit_transform**(*self*, *X*, *y=None*)

Fits on X and transforms X.

**Parameters**

- **X** (*pd.DataFrame*) – Data to fit and transform.
- **y** (*pd.Series*) – Target data.

**Returns** Transformed X.

**Return type** pd.DataFrame

**Raises** **MethodPropertyNotFoundError** – If transformer does not have a transform method or a component_obj that implements transform.

**static load**(*file_path*)

Loads component at file path.

**Parameters** **file_path** (*str*) – Location to load file.

**Returns** ComponentBase object

**needs_fitting**(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need to be fit or whose fit methods do nothing.

**Returns** True.

**property parameters**(*self*)

Returns the parameters which were used to initialize the component.

**save**(*self*, *file_path*, *pickle_protocol=cloudpickle.DEFAULT_PROTOCOL*)

Saves component at file path.

**Parameters**

- **file_path** (*str*) – Location to save file.
- **pickle_protocol** (*int*) – The pickle data stream format.

**transform**(*self*, *X*, *y=None*)

Transforms data X by creating new features using existing text columns.

**Parameters**

- **X** (*pd.DataFrame*) – The data to transform.
- **y** (*pd.Series, optional*) – Ignored.

**Returns** Transformed X

**Return type** pd.DataFrame

**update_parameters**(*self*, *update_dict*, *reset_fit=True*)

Updates the parameter dictionary of the component.

**Parameters**

- **update_dict** (*dict*) – A dict of parameters to update.
- **reset_fit** (*bool, optional*) – If True, will set *_is_fitted* to False.

**class** evalml.pipelines.components.transformers.preprocessing.**PolynomialDecomposer**(*time_index: str = None, degree: int = 1, period: int = - 1, random_seed: int = 0, **kwargs*)

Removes trends and seasonality from time series by fitting a polynomial and moving average to the data.

**Scikit-learn's PolynomialForecaster is used to generate the additive trend portion of the target data. A polynomial** will be fit to the data during fit. That additive polynomial trend will be removed during fit so that statsmodel's seasonal_decompose can determine the addititve seasonality of the data by using rolling averages over the series' inferred periodicity.

For example, daily time series data will generate rolling averages over the first week of data, normalize out the mean and return those 7 averages repeated over the entire length of the given series. Those seven averages, repeated as many times as necessary to match the length of the given target data, will be used as the seasonal signal of the data.

**Parameters**

- **time_index** (*str*) – Specifies the name of the column in X that provides the datetime objects. Defaults to None.
- **degree** (*int*) – Degree for the polynomial. If 1, linear model is fit to the data. If 2, quadratic model is fit, etc. Defaults to 1.

- **period** (*int*) – The number of entries in the time series data that corresponds to one period of a cyclic signal. For instance, if data is known to possess a weekly seasonal signal, and if the data is daily data, period should be 7. For daily data with a yearly seasonal signal, period should be 365. Defaults to -1, which uses the statsmodels libarary's freq_to_period function. https://github.com/statsmodels/statsmodels/blob/main/statsmodels/tsa/tsatools.py
- **random_seed** (*int*) – Seed for the random number generator. Defaults to 0.

**Attributes**

| | |
|---|---|
| **hyper-parameter_ranges** | { "degree": Integer(1, 3)} |
| **invalid_frequencies** | [] |
| **modifies_features** | False |
| **modifies_target** | True |
| **name** | Polynomial Decomposer |
| **needs_fitting** | True |
| **training_only** | False |

**Methods**

| | |
|---|---|
| *clone* | Constructs a new component with the same parameters and random state. |
| *default_parameters* | Returns the default parameters for this component. |
| *describe* | Describe a component and its parameters. |
| *determine_periodicity* | Function that uses autocorrelative methods to determine the likely most signficant period of the seasonal signal. |
| *fit* | Fits the PolynomialDecomposer and determine the seasonal signal. |
| *fit_transform* | Removes fitted trend and seasonality from target variable. |
| *get_trend_dataframe* | Return a list of dataframes with 4 columns: signal, trend, seasonality, residual. |
| *inverse_transform* | Adds back fitted trend and seasonality to target variable. |
| *is_freq_valid* | Determines if the given string represents a valid frequency for this decomposer. |
| *load* | Loads component at file path. |
| *parameters* | Returns the parameters which were used to initialize the component. |
| *plot_decomposition* | Plots the decomposition of the target signal. |
| *save* | Saves component at file path. |
| *set_period* | Function to set the component's seasonal period based on the target's seasonality. |
| *transform* | Transforms the target data by removing the polynomial trend and rolling average seasonality. |
| *update_parameters* | Updates the parameter dictionary of the component. |

**clone**(*self*)

Constructs a new component with the same parameters and random state.

**Returns** A new instance of this component with identical parameters and random state.

**default_parameters**(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

**Returns** Default parameters for this component.

**Return type** dict

**describe**(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

**Parameters**

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description as dictionary in the format {"name": name, "parameters": parameters}

**Returns** Returns dictionary if return_dict is True, else None.

**Return type** None or dict

**classmethod determine_periodicity**(*cls*, *X: pandas.DataFrame*, *y: pandas.Series*, *acf_threshold: float = 0.01*, *rel_max_order: int = 5*)

Function that uses autocorrelative methods to determine the likely most signficant period of the seasonal signal.

**Parameters**

- **X** (*pandas.DataFrame*) – The feature data of the time series problem.
- **y** (*pandas.Series*) – The target data of a time series problem.
- **acf_threshold** (*float*) – The threshold for the autocorrelation function to determine the period. Any values below the threshold are considered to be 0 and will not be considered for the period. Defaults to 0.01.
- **rel_max_order** (*int*) – The order of the relative maximum to determine the period. Defaults to 5.

**Returns**

**The integer number of entries in time series data over which the seasonal part of the target data** repeats. If the time series data is in days, then this is the number of days that it takes the target's seasonal signal to repeat. Note: the target data can contain multiple seasonal signals. This function will only return the stronger. E.g. if the target has both weekly and yearly seasonality, the function may return either "7" or "365", depending on which seasonality is more strongly autocorrelated. If no period is detected, returns None.

**Return type** int

**fit**(*self*, *X: pandas.DataFrame*, *y: pandas.Series = None*) → *PolynomialDecomposer*

Fits the PolynomialDecomposer and determine the seasonal signal.

Currently only fits the polynomial detrender. The seasonality is determined by removing the trend from the signal and using statsmodels' seasonal_decompose(). Both the trend and seasonality are currently assumed to be additive.

**Parameters**

- **X** (*pd.DataFrame, optional*) – Conditionally used to build datetime index.
- **y** (*pd.Series*) – Target variable to detrend and deseasonalize.

**Returns** self

**Raises**

- **NotImplementedError** – If the input data has a frequency of "month-begin". This isn't supported by statsmodels decompose as the freqstr "MS" is misinterpreted as milliseconds.
- **ValueError** – If y is None.
- **ValueError** – If target data doesn't have DatetimeIndex AND no Datetime features in features data

**fit_transform**(*self*, *X: pandas.DataFrame*, *y: pandas.Series = None*) → tuple[pandas.DataFrame, pandas.Series]

Removes fitted trend and seasonality from target variable.

**Parameters**

- **X** (*pd.DataFrame, optional*) – Ignored.
- **y** (*pd.Series*) – Target variable to detrend and deseasonalize.

**Returns**

**The first element are the input features returned without modification.** The second element is the target variable y with the fitted trend removed.

**Return type** tuple of pd.DataFrame, pd.Series

**get_trend_dataframe**(*self*, *X: pandas.DataFrame*, *y: pandas.Series*) → list[pandas.DataFrame]

Return a list of dataframes with 4 columns: signal, trend, seasonality, residual.

Scikit-learn's PolynomialForecaster is used to generate the trend portion of the target data. statsmodel's seasonal_decompose is used to generate the seasonality of the data.

**Parameters**

- **X** (*pd.DataFrame*) – Input data with time series data in index.
- **y** (*pd.Series or pd.DataFrame*) – Target variable data provided as a Series for univariate problems or a DataFrame for multivariate problems.

**Returns**

**Each DataFrame contains the columns "signal", "trend", "seasonality" and "residual,"** with the latter 3 column values being the decomposed elements of the target data. The "signal" column is simply the input target signal but reindexed with a datetime index to match the input features.

**Return type** list of pd.DataFrame

**Raises**

- **TypeError** – If X does not have time-series data in the index.
- **ValueError** – If time series index of X does not have an inferred frequency.
- **ValueError** – If the forecaster associated with the detrender has not been fit yet.
- **TypeError** – If y is not provided as a pandas Series or DataFrame.

**inverse_transform**(*self*, *y_t: pandas.Series*) → tuple[pandas.DataFrame, pandas.Series]

Adds back fitted trend and seasonality to target variable.

The polynomial trend is added back into the signal, calling the detrender's inverse_transform(). Then, the seasonality is projected forward to and added back into the signal.

**Parameters** **y_t** (*pd.Series*) – Target variable.

**Returns**

**The first element are the input features returned without modification.** The second element is the target variable y with the trend and seasonality added back in.

**Return type** tuple of pd.DataFrame, pd.Series

**Raises** **ValueError** – If y is None.

**classmethod is_freq_valid**(*cls*, *freq: str*)

Determines if the given string represents a valid frequency for this decomposer.

**Parameters** **freq** (*str*) – A frequency to validate. See the pandas docs at https://pandas.pydata.org/pandas-docs/stable/user_guide/timeseries.html#offset-aliases for options.

**Returns** boolean representing whether the frequency is valid or not.

**static load**(*file_path*)

Loads component at file path.

**Parameters** **file_path** (*str*) – Location to load file.

**Returns** ComponentBase object

**property parameters**(*self*)

Returns the parameters which were used to initialize the component.

**plot_decomposition**(*self*, *X: pandas.DataFrame*, *y: pandas.Series*, *show: bool = False*) → tuple[matplotlib.pyplot.Figure, list]

Plots the decomposition of the target signal.

**Parameters**

- **X** (*pd.DataFrame*) – Input data with time series data in index.
- **y** (*pd.Series or pd.DataFrame*) – Target variable data provided as a Series for univariate problems or a DataFrame for multivariate problems.
- **show** (*bool*) – Whether to display the plot or not. Defaults to False.

**Returns**

**The figure and axes that have the decompositions** plotted on them

**Return type** matplotlib.pyplot.Figure, list[matplotlib.pyplot.Axes]

**save**(*self*, *file_path*, *pickle_protocol=cloudpickle.DEFAULT_PROTOCOL*)

Saves component at file path.

**Parameters**

- **file_path** (*str*) – Location to save file.
- **pickle_protocol** (*int*) – The pickle data stream format.

**set_period**(*self*, *X: pandas.DataFrame*, *y: pandas.Series*, *acf_threshold: float = 0.01*, *rel_max_order: int = 5*)

Function to set the component's seasonal period based on the target's seasonality.

**Parameters**

- **X** (*pandas.DataFrame*) – The feature data of the time series problem.
- **y** (*pandas.Series*) – The target data of a time series problem.
- **acf_threshold** (*float*) – The threshold for the autocorrelation function to determine the period. Any values below the threshold are considered to be 0 and will not be considered for the period. Defaults to 0.01.
- **rel_max_order** (*int*) – The order of the relative maximum to determine the period. Defaults to 5.

**transform**(*self*, *X: pandas.DataFrame*, *y: pandas.Series = None*) → tuple[pandas.DataFrame, pandas.Series]

Transforms the target data by removing the polynomial trend and rolling average seasonality.

Applies the fit polynomial detrender to the target data, removing the additive polynomial trend. Then, utilizes the first period's worth of seasonal data determined in the .fit() function to extrapolate the seasonal signal of the data to be transformed. This seasonal signal is also assumed to be additive and is removed.

**Parameters**

- **X** (*pd.DataFrame, optional*) – Conditionally used to build datetime index.
- **y** (*pd.Series*) – Target variable to detrend and deseasonalize.

**Returns**

**The input features are returned without modification. The target** variable y is detrended and deseasonalized.

**Return type** tuple of pd.DataFrame, pd.Series

**Raises** **ValueError** – If target data doesn't have DatetimeIndex AND no Datetime features in features data

**update_parameters**(*self*, *update_dict*, *reset_fit=True*)

Updates the parameter dictionary of the component.

**Parameters**

- **update_dict** (*dict*) – A dict of parameters to update.
- **reset_fit** (*bool, optional*) – If True, will set *_is_fitted* to False.

**class** evalml.pipelines.components.transformers.preprocessing.**ReplaceNullableTypes**(*random_seed=0*, ***kwargs*)

Transformer to replace features with the new nullable dtypes with a dtype that is compatible in EvalML.

**Attributes**

| | |
|---|---|
| **hyper-parameter_ranges** | None |
| **modifies_features** | True |
| **modifies_target** | {} |
| **name** | Replace Nullable Types Transformer |
| **training_only** | False |

**Methods**

| | |
|---|---|
| *clone* | Constructs a new component with the same parameters and random state. |
| *default_parameters* | Returns the default parameters for this component. |
| *describe* | Describe a component and its parameters. |
| *fit* | Fits component to data. |
| *fit_transform* | Substitutes non-nullable types for the new pandas nullable types in the data and target data. |
| *load* | Loads component at file path. |
| *needs_fitting* | Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances. |
| *parameters* | Returns the parameters which were used to initialize the component. |
| *save* | Saves component at file path. |
| *transform* | Transforms data by replacing columns that contain nullable types with the appropriate replacement type. |
| *update_parameters* | Updates the parameter dictionary of the component. |

**clone**(*self*)

Constructs a new component with the same parameters and random state.

**Returns** A new instance of this component with identical parameters and random state.

**default_parameters**(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

**Returns** Default parameters for this component.

**Return type** dict

**describe**(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

**Parameters**

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description as dictionary in the format {"name": name, "parameters": parameters}

**Returns** Returns dictionary if return_dict is True, else None.

**Return type** None or dict

**fit**(*self*, *X*, *y=None*)

Fits component to data.

**Parameters**

- **X** (*pd.DataFrame*) – The input training data of shape [n_samples, n_features].
- **y** (*pd.Series, optional*) – The target training data of length [n_samples].

**Returns** self

**fit_transform**(*self*, *X*, *y=None*)

Substitutes non-nullable types for the new pandas nullable types in the data and target data.

**Parameters**

- **X** (*pd.DataFrame, optional*) – Input features.
- **y** (*pd.Series*) – Target data.

**Returns** The input features and target data with the non-nullable types set.

**Return type** tuple of pd.DataFrame, pd.Series

**static load**(*file_path*)

Loads component at file path.

**Parameters** **file_path** (*str*) – Location to load file.

**Returns** ComponentBase object

**needs_fitting**(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need to be fit or whose fit methods do nothing.

**Returns** True.

**property parameters**(*self*)

Returns the parameters which were used to initialize the component.

**save**(*self*, *file_path*, *pickle_protocol=cloudpickle.DEFAULT_PROTOCOL*)

Saves component at file path.

**Parameters**

- **file_path** (*str*) – Location to save file.
- **pickle_protocol** (*int*) – The pickle data stream format.

**transform**(*self*, *X*, *y=None*)

Transforms data by replacing columns that contain nullable types with the appropriate replacement type.

"float64" for nullable integers and "category" for nullable booleans.

**Parameters**

- **X** (*pd.DataFrame*) – Data to transform
- **y** (*pd.Series, optional*) – Target data to transform

**Returns** Transformed X pd.Series: Transformed y

**Return type** pd.DataFrame

**update_parameters**(*self*, *update_dict*, *reset_fit=True*)

Updates the parameter dictionary of the component.

**Parameters**

- **update_dict** (*dict*) – A dict of parameters to update.
- **reset_fit** (*bool, optional*) – If True, will set *_is_fitted* to False.

**class** evalml.pipelines.components.transformers.preprocessing.**STLDecomposer**(*time_index: str = None*, *degree: int = 1*, *period: int = None*, *seasonal_smoother: int = 7*, *random_seed: int = 0*, ***kwargs*)

Removes trends and seasonality from time series using the STL algorithm.

https://www.statsmodels.org/dev/generated/statsmodels.tsa.seasonal.STL.html

**Parameters**

- **time_index** (*str*) – Specifies the name of the column in X that provides the datetime objects. Defaults to None.
- **degree** (*int*) – Not currently used. STL 3x "degree-like" values. None are able to be set at this time. Defaults to 1.
- **period** (*int*) – The number of entries in the time series data that corresponds to one period of a cyclic signal. For instance, if data is known to possess a weekly seasonal signal, and if the data is daily data, the period should likely be 7. For daily data with a yearly seasonal signal, the period should likely be 365. If None, statsmodels will infer the period based on the frequency. Defaults to None.
- **seasonal_smoother** (*int*) – The length of the seasonal smoother used by the underlying STL algorithm. For compatibility, must be odd. If an even number is provided, the next, highest odd number will be used. Defaults to 7.
- **random_seed** (*int*) – Seed for the random number generator. Defaults to 0.

**Attributes**

| | |
|---|---|
| **hyper-parameter_ranges** | None |
| **invalid_frequencies** | [] |
| **modifies_features** | False |
| **modifies_target** | True |
| **name** | STL Decomposer |
| **needs_fitting** | True |
| **training_only** | False |

**Methods**

| | |
|---|---|
| clone | Constructs a new component with the same parameters and random state. |
| default_parameters | Returns the default parameters for this component. |
| describe | Describe a component and its parameters. |
| determine_periodicity | Function that uses autocorrelative methods to determine the likely most signficant period of the seasonal signal. |
| fit | Fits the STLDecomposer and determine the seasonal signal. |
| fit_transform | Removes fitted trend and seasonality from target variable. |
| get_trend_dataframe | Return a list of dataframes with 4 columns: signal, trend, seasonality, residual. |
| get_trend_prediction_intervals | Calculate the prediction intervals for the trend data. |
| inverse_transform | Adds back fitted trend and seasonality to target variable. |
| is_freq_valid | Determines if the given string represents a valid frequency for this decomposer. |
| load | Loads component at file path. |
| parameters | Returns the parameters which were used to initialize the component. |
| plot_decomposition | Plots the decomposition of the target signal. |
| save | Saves component at file path. |
| set_period | Function to set the component's seasonal period based on the target's seasonality. |
| transform | Transforms the target data by removing the STL trend and seasonality. |
| update_parameters | Updates the parameter dictionary of the component. |

**clone**(*self*)

Constructs a new component with the same parameters and random state.

**Returns** A new instance of this component with identical parameters and random state.

**default_parameters**(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

**Returns** Default parameters for this component.

**Return type** dict

**describe**(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

**Parameters**

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description as dictionary in the format {"name": name, "parameters": parameters}

**Returns** Returns dictionary if return_dict is True, else None.

**Return type** None or dict

**classmethod determine_periodicity**(*cls*, *X: pandas.DataFrame*, *y: pandas.Series*, *acf_threshold: float = 0.01*, *rel_max_order: int = 5*)

Function that uses autocorrelative methods to determine the likely most signficant period of the seasonal signal.

**Parameters**

- **X** (*pandas.DataFrame*) – The feature data of the time series problem.
- **y** (*pandas.Series*) – The target data of a time series problem.
- **acf_threshold** (*float*) – The threshold for the autocorrelation function to determine the period. Any values below the threshold are considered to be 0 and will not be considered for the period. Defaults to 0.01.
- **rel_max_order** (*int*) – The order of the relative maximum to determine the period. Defaults to 5.

**Returns**

**The integer number of entries in time series data over which the seasonal part of the target data** repeats. If the time series data is in days, then this is the number of days that it takes the target's seasonal signal to repeat. Note: the target data can contain multiple seasonal signals. This function will only return the stronger. E.g. if the target has both weekly and yearly seasonality, the function may return either "7" or "365", depending on which seasonality is more strongly autocorrelated. If no period is detected, returns None.

**Return type** int

**fit**(*self*, *X: pandas.DataFrame*, *y: pandas.Series = None*) → *STLDecomposer*

Fits the STLDecomposer and determine the seasonal signal.

Instantiates a statsmodels STL decompose object with the component's stored parameters and fits it. Since the statsmodels object does not fit the sklearn api, it is not saved during __init__() in _component_obj and will be re-instantiated each time fit is called.

To emulate the sklearn API, when the STL decomposer is fit, the full seasonal component, a single period sample of the seasonal component, the full trend-cycle component and the residual are saved.

y(t) = S(t) + T(t) + R(t)

**Parameters**

- **X** (*pd.DataFrame, optional*) – Conditionally used to build datetime index.
- **y** (*pd.Series*) – Target variable to detrend and deseasonalize.

**Returns** self

**Raises**

- **ValueError** – If y is None.
- **ValueError** – If target data doesn't have DatetimeIndex AND no Datetime features in features data

**fit_transform**(*self*, *X: pandas.DataFrame*, *y: pandas.Series = None*) → tuple[pandas.DataFrame, pandas.Series]

Removes fitted trend and seasonality from target variable.

**Parameters**

- **X** (*pd.DataFrame, optional*) – Ignored.
- **y** (*pd.Series*) – Target variable to detrend and deseasonalize.

**Returns**

**The first element are the input features returned without modification.** The second element is the target variable y with the fitted trend removed.

**Return type** tuple of pd.DataFrame, pd.Series

`get_trend_dataframe`(*self*, *X*, *y*)

Return a list of dataframes with 4 columns: signal, trend, seasonality, residual.

**Parameters**

- **X** (*pd.DataFrame*) – Input data with time series data in index.
- **y** (*pd.Series or pd.DataFrame*) – Target variable data provided as a Series for univariate problems or a DataFrame for multivariate problems.

**Returns**

**Each DataFrame contains the columns "signal", "trend", "seasonality" and "residual,"** with the latter 3 column values being the decomposed elements of the target data. The "signal" column is simply the input target signal but reindexed with a datetime index to match the input features.

**Return type** list of pd.DataFrame

**Raises**

- **TypeError** – If X does not have time-series data in the index.
- **ValueError** – If time series index of X does not have an inferred frequency.
- **ValueError** – If the forecaster associated with the detrender has not been fit yet.
- **TypeError** – If y is not provided as a pandas Series or DataFrame.

`get_trend_prediction_intervals`(*self*, *y*, *coverage=None*)

Calculate the prediction intervals for the trend data.

**Parameters**

- **y** (*pd.Series*) – Target data.
- **coverage** (*list[float]*) – A list of floats between the values 0 and 1 that the upper and lower bounds of the prediction interval should be calculated for.

**Returns** Prediction intervals, keys are in the format {coverage}_lower or {coverage}_upper.

**Return type** dict of pd.Series

`inverse_transform`(*self*, *y_t: pandas.Series*) → tuple[pandas.DataFrame, pandas.Series]

Adds back fitted trend and seasonality to target variable.

The STL trend is projected to cover the entire requested target range, then added back into the signal. Then, the seasonality is projected forward to and added back into the signal.

**Parameters** **y_t** (*pd.Series*) – Target variable.

**Returns**

**The first element are the input features returned without modification.** The second element is the target variable y with the trend and seasonality added back in.

**Return type** tuple of pd.DataFrame, pd.Series

**Raises** **ValueError** – If y is None.

**classmethod is_freq_valid**(*cls*, *freq: str*)

Determines if the given string represents a valid frequency for this decomposer.

**Parameters** **freq** (*str*) – A frequency to validate. See the pandas docs at https://pandas.pydata.org/pandas-docs/stable/user_guide/timeseries.html#offset-aliases for options.

**Returns** boolean representing whether the frequency is valid or not.

**static load**(*file_path*)

Loads component at file path.

**Parameters** **file_path** (*str*) – Location to load file.

**Returns** ComponentBase object

**property parameters**(*self*)

Returns the parameters which were used to initialize the component.

**plot_decomposition**(*self*, *X: pandas.DataFrame*, *y: pandas.Series*, *show: bool = False*) → tuple[matplotlib.pyplot.Figure, list]

Plots the decomposition of the target signal.

**Parameters**

- **X** (*pd.DataFrame*) – Input data with time series data in index.
- **y** (*pd.Series or pd.DataFrame*) – Target variable data provided as a Series for univariate problems or a DataFrame for multivariate problems.
- **show** (*bool*) – Whether to display the plot or not. Defaults to False.

**Returns**

**The figure and axes that have the decompositions** plotted on them

**Return type** matplotlib.pyplot.Figure, list[matplotlib.pyplot.Axes]

**save**(*self*, *file_path*, *pickle_protocol=cloudpickle.DEFAULT_PROTOCOL*)

Saves component at file path.

**Parameters**

- **file_path** (*str*) – Location to save file.
- **pickle_protocol** (*int*) – The pickle data stream format.

**set_period**(*self*, *X: pandas.DataFrame*, *y: pandas.Series*, *acf_threshold: float = 0.01*, *rel_max_order: int = 5*)

Function to set the component's seasonal period based on the target's seasonality.

**Parameters**

- **X** (*pandas.DataFrame*) – The feature data of the time series problem.
- **y** (*pandas.Series*) – The target data of a time series problem.
- **acf_threshold** (*float*) – The threshold for the autocorrelation function to determine the period. Any values below the threshold are considered to be 0 and will not be considered for the period. Defaults to 0.01.
- **rel_max_order** (*int*) – The order of the relative maximum to determine the period. Defaults to 5.

**transform**(*self*, *X: pandas.DataFrame*, *y: pandas.Series = None*) → tuple[pandas.DataFrame, pandas.Series]

Transforms the target data by removing the STL trend and seasonality.

Uses an ARIMA model to project forward the addititve trend and removes it. Then, utilizes the first period's worth of seasonal data determined in the .fit() function to extrapolate the seasonal signal of the data to be transformed. This seasonal signal is also assumed to be additive and is removed.

**Parameters**

- **X** (*pd.DataFrame, optional*) – Conditionally used to build datetime index.
- **y** (*pd.Series*) – Target variable to detrend and deseasonalize.

**Returns**

**The input features are returned without modification. The target** variable y is detrended and deseasonalized.

**Return type** tuple of pd.DataFrame, pd.Series

**Raises** **ValueError** – If target data doesn't have DatetimeIndex AND no Datetime features in features data

**update_parameters**(*self*, *update_dict*, *reset_fit=True*)

Updates the parameter dictionary of the component.

**Parameters**

- **update_dict** (*dict*) – A dict of parameters to update.
- **reset_fit** (*bool, optional*) – If True, will set *_is_fitted* to False.

**class** evalml.pipelines.components.transformers.preprocessing.**TextTransformer**(*component_obj=None*, *random_seed=0*, ***kwargs*)

Base class for all transformers working with text features.

**Parameters**

- **component_obj** (*obj*) – Third-party objects useful in component implementation. Defaults to None.
- **random_seed** (*int*) – Seed for the random number generator. Defaults to 0.

**Attributes**

| | |
|---|---|
| **modifies_features** | True |
| **modifies_target** | False |
| **training_only** | False |

**Methods**

| | |
|---|---|
| *clone* | Constructs a new component with the same parameters and random state. |
| *default_parameters* | Returns the default parameters for this component. |
| *describe* | Describe a component and its parameters. |
| *fit* | Fits component to data. |
| *fit_transform* | Fits on X and transforms X. |
| *load* | Loads component at file path. |
| *name* | Returns string name of this component. |
| *needs_fitting* | Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances. |
| *parameters* | Returns the parameters which were used to initialize the component. |
| *save* | Saves component at file path. |
| *transform* | Transforms data X. |
| *update_parameters* | Updates the parameter dictionary of the component. |

**clone**(*self*)

Constructs a new component with the same parameters and random state.

**Returns** A new instance of this component with identical parameters and random state.

**default_parameters**(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

**Returns** Default parameters for this component.

**Return type** dict

**describe**(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

**Parameters**

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description as dictionary in the format {"name": name, "parameters": parameters}

**Returns** Returns dictionary if return_dict is True, else None.

**Return type** None or dict

**fit**(*self*, *X*, *y=None*)

Fits component to data.

**Parameters**

- **X** (*pd.DataFrame*) – The input training data of shape [n_samples, n_features]
- **y** (*pd.Series, optional*) – The target training data of length [n_samples]

**Returns** self

**Raises** **MethodPropertyNotFoundError** – If component does not have a fit method or a component_obj that implements fit.

**fit_transform**(*self*, *X*, *y=None*)

Fits on X and transforms X.

**Parameters**

- **X** (*pd.DataFrame*) – Data to fit and transform.
- **y** (*pd.Series*) – Target data.

**Returns** Transformed X.

**Return type** pd.DataFrame

**Raises** **MethodPropertyNotFoundError** – If transformer does not have a transform method or a component_obj that implements transform.

**static load**(*file_path*)

Loads component at file path.

**Parameters** **file_path** (*str*) – Location to load file.

**Returns** ComponentBase object

**property name**(*cls*)

Returns string name of this component.

**needs_fitting**(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need to be fit or whose fit methods do nothing.

**Returns** True.

**property parameters**(*self*)

Returns the parameters which were used to initialize the component.

**save**(*self*, *file_path*, *pickle_protocol=cloudpickle.DEFAULT_PROTOCOL*)

Saves component at file path.

**Parameters**

- **file_path** (*str*) – Location to save file.
- **pickle_protocol** (*int*) – The pickle data stream format.

**abstract transform**(*self*, *X*, *y=None*)

Transforms data X.

**Parameters**

- **X** (*pd.DataFrame*) – Data to transform.
- **y** (*pd.Series, optional*) – Target data.

**Returns** Transformed X

**Return type** pd.DataFrame

**Raises** **MethodPropertyNotFoundError** – If transformer does not have a transform method or a component_obj that implements transform.

**update_parameters**(*self*, *update_dict*, *reset_fit=True*)

Updates the parameter dictionary of the component.

**Parameters**

- **update_dict** (*dict*) – A dict of parameters to update.
- **reset_fit** (*bool, optional*) – If True, will set *_is_fitted* to False.

**class** evalml.pipelines.components.transformers.preprocessing.**TimeSeriesFeaturizer**(*time_index=None*, *max_delay=2*, *gap=0*, *forecast_horizon=1*, *conf_level=0.05*, *rolling_window_size=0.25*, *delay_features=True*, *delay_target=True*, *random_seed=0*, ***kwargs*)

Transformer that delays input features and target variable for time series problems.

This component uses an algorithm based on the autocorrelation values of the target variable to determine which lags to select from the set of all possible lags.

The algorithm is based on the idea that the local maxima of the autocorrelation function indicate the lags that have the most impact on the present time.

The algorithm computes the autocorrelation values and finds the local maxima, called "peaks", that are significant at the given conf_level. Since lags in the range [0, 10] tend to be predictive but not local maxima, the union of the peaks is taken with the significant lags in the range [0, 10]. At the end, only selected lags in the range [0, max_delay] are used.

Parametrizing the algorithm by conf_level lets the AutoMLAlgorithm tune the set of lags chosen so that the chances of finding a good set of lags is higher.

Using conf_level value of 1 selects all possible lags.

**Parameters**

- **time_index** (*str*) – Name of the column containing the datetime information used to order the data. Ignored.
- **max_delay** (*int*) – Maximum number of time units to delay each feature. Defaults to 2.
- **forecast_horizon** (*int*) – The number of time periods the pipeline is expected to forecast.
- **conf_level** (*float*) – Float in range (0, 1] that determines the confidence interval size used to select which lags to compute from the set of [1, max_delay]. A delay of 1 will always be computed. If 1, selects all possible lags in the set of [1, max_delay], inclusive.
- **rolling_window_size** (*float*) – Float in range (0, 1] that determines the size of the window used for rolling features. Size is computed as rolling_window_size * max_delay.
- **delay_features** (*bool*) – Whether to delay the input features. Defaults to True.
- **delay_target** (*bool*) – Whether to delay the target. Defaults to True.

- **gap** (*int*) – The number of time units between when the features are collected and when the target is collected. For example, if you are predicting the next time step's target, gap=1. This is only needed because when gap=0, we need to be sure to start the lagging of the target variable at 1. Defaults to 1.
- **random_seed** (*int*) – Seed for the random number generator. This transformer performs the same regardless of the random seed provided.

**Attributes**

| | |
|---|---|
| **hyper-parameter_ranges** | Real(0.001, 1.0), "rolling_window_size": Real(0.001, 1.0)}:type: {"conf_level" |
| **modifies_features** | True |
| **modifies_target** | False |
| **name** | Time Series Featurizer |
| **needs_fitting** | True |
| **target_colname_prefix** | target_delay_{} |
| **training_only** | False |

**Methods**

| | |
|---|---|
| *clone* | Constructs a new component with the same parameters and random state. |
| *default_parameters* | Returns the default parameters for this component. |
| *describe* | Describe a component and its parameters. |
| *fit* | Fits the DelayFeatureTransformer. |
| *fit_transform* | Fit the component and transform the input data. |
| *load* | Loads component at file path. |
| *parameters* | Returns the parameters which were used to initialize the component. |
| *save* | Saves component at file path. |
| *transform* | Computes the delayed values and rolling means for X and y. |
| *update_parameters* | Updates the parameter dictionary of the component. |

**clone**(*self*)

Constructs a new component with the same parameters and random state.

**Returns** A new instance of this component with identical parameters and random state.

**default_parameters**(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

**Returns** Default parameters for this component.

**Return type** dict

**describe**(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

**Parameters**

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description as dictionary in the format {"name": name, "parameters": parameters}

**Returns** Returns dictionary if return_dict is True, else None.

**Return type** None or dict

**fit**(*self*, *X*, *y=None*)

Fits the DelayFeatureTransformer.

**Parameters**

- **X** (*pd.DataFrame or np.ndarray*) – The input training data of shape [n_samples, n_features]
- **y** (*pd.Series, optional*) – The target training data of length [n_samples]

**Returns** self

**Raises** **ValueError** – if self.time_index is None

**fit_transform**(*self*, *X*, *y=None*)

Fit the component and transform the input data.

**Parameters**

- **X** (*pd.DataFrame*) – Data to transform.
- **y** (*pd.Series, or None*) – Target.

**Returns** Transformed X.

**Return type** pd.DataFrame

**static load**(*file_path*)

Loads component at file path.

**Parameters** **file_path** (*str*) – Location to load file.

**Returns** ComponentBase object

**property parameters**(*self*)

Returns the parameters which were used to initialize the component.

**save**(*self*, *file_path*, *pickle_protocol=cloudpickle.DEFAULT_PROTOCOL*)

Saves component at file path.

**Parameters**

- **file_path** (*str*) – Location to save file.
- **pickle_protocol** (*int*) – The pickle data stream format.

**transform**(*self*, *X*, *y=None*)

Computes the delayed values and rolling means for X and y.

The chosen delays are determined by the autocorrelation function of the target variable. See the class docstring for more information on how they are chosen. If y is None, all possible lags are chosen.

If y is not None, it will also compute the delayed values for the target variable.

The rolling means for all numeric features in X and y, if y is numeric, are also returned.

**Parameters**

- **X** (*pd.DataFrame or None*) – Data to transform. None is expected when only the target variable is being used.
- **y** (*pd.Series, or None*) – Target.

**Returns** Transformed X. No original features are returned.

**Return type** pd.DataFrame

**update_parameters**(*self*, *update_dict*, *reset_fit=True*)

Updates the parameter dictionary of the component.

**Parameters**

- **update_dict** (*dict*) – A dict of parameters to update.
- **reset_fit** (*bool, optional*) – If True, will set *_is_fitted* to False.

class evalml.pipelines.components.transformers.preprocessing.**TimeSeriesRegularizer**(*time_index=None*, *frequency_payload=None*, *window_length=4*, *threshold=0.4*, *random_seed=0*, ***kwargs*)

Transformer that regularizes an inconsistently spaced datetime column.

If X is passed in to fit/transform, the column *time_index* will be checked for an inferrable offset frequency. If the *time_index* column is perfectly inferrable then this Transformer will do nothing and return the original X and y.

If X does not have a perfectly inferrable frequency but one can be estimated, then X and y will be reformatted based on the estimated frequency for *time_index*. In the original X and y passed: - Missing datetime values will be added and will have their corresponding columns in X and y set to None. - Duplicate datetime values will be dropped. - Extra datetime values will be dropped. - If it can be determined that a duplicate or extra value is misaligned, then it will be repositioned to take the place of a missing value.

This Transformer should be used before the *TimeSeriesImputer* in order to impute the missing values that were added to X and y (if passed).

**Parameters**

- **time_index** (*string*) – Name of the column containing the datetime information used to order the data, required. Defaults to None.
- **frequency_payload** (*tuple*) – Payload returned from Woodwork's infer_frequency function where debug is True. Defaults to None.
- **window_length** (*int*) – The size of the rolling window over which inference is conducted to determine the prevalence of uninferrable frequencies.
- **5.** (*Lower values make this component more sensitive to recognizing numerous faulty datetime values. Defaults to*) –
- **threshold** (*float*) – The minimum percentage of windows that need to have been able to infer a frequency. Lower values make this component more
- **0.8.** (*sensitive to recognizing numerous faulty datetime values. Defaults to*) –

- **random_seed** (*int*) – Seed for the random number generator. This transformer performs the same regardless of the random seed provided.
- **0.** (*Defaults to*) –

**Raises** **ValueError** – if the frequency_payload parameter has not been passed a tuple

**Attributes**

| | |
|---|---|
| **hyper-parameter_ranges** | {} |
| **modifies_features** | True |
| **modifies_target** | True |
| **name** | Time Series Regularizer |
| **training_only** | True |

**Methods**

| | |
|---|---|
| *clone* | Constructs a new component with the same parameters and random state. |
| *default_parameters* | Returns the default parameters for this component. |
| *describe* | Describe a component and its parameters. |
| *fit* | Fits the TimeSeriesRegularizer. |
| *fit_transform* | Fits on X and transforms X. |
| *load* | Loads component at file path. |
| *needs_fitting* | Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances. |
| *parameters* | Returns the parameters which were used to initialize the component. |
| *save* | Saves component at file path. |
| *transform* | Regularizes a dataframe and target data to an inferrable offset frequency. |
| *update_parameters* | Updates the parameter dictionary of the component. |

**clone**(*self*)

Constructs a new component with the same parameters and random state.

**Returns** A new instance of this component with identical parameters and random state.

**default_parameters**(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

**Returns** Default parameters for this component.

**Return type** dict

**describe**(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

**Parameters**

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description as dictionary in the format {"name": name, "parameters": parameters}

**Returns** Returns dictionary if return_dict is True, else None.

**Return type** None or dict

**fit**(*self*, *X*, *y=None*)

Fits the TimeSeriesRegularizer.

**Parameters**

- **X** (*pd.DataFrame*) – The input training data of shape [n_samples, n_features].
- **y** (*pd.Series, optional*) – The target training data of length [n_samples].

**Returns** self

**Raises**

- **ValueError** – if self.time_index is None, if X and y have different lengths, if *time_index* in X does not have an offset frequency that can be estimated
- **TypeError** – if the *time_index* column is not of type Datetime
- **KeyError** – if the *time_index* column doesn't exist

**fit_transform**(*self*, *X*, *y=None*)

Fits on X and transforms X.

**Parameters**

- **X** (*pd.DataFrame*) – Data to fit and transform.
- **y** (*pd.Series*) – Target data.

**Returns** Transformed X.

**Return type** pd.DataFrame

**Raises** **MethodPropertyNotFoundError** – If transformer does not have a transform method or a component_obj that implements transform.

**static load**(*file_path*)

Loads component at file path.

**Parameters** **file_path** (*str*) – Location to load file.

**Returns** ComponentBase object

**needs_fitting**(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need to be fit or whose fit methods do nothing.

**Returns** True.

**property parameters**(*self*)

Returns the parameters which were used to initialize the component.

**save**(*self*, *file_path*, *pickle_protocol=cloudpickle.DEFAULT_PROTOCOL*)

Saves component at file path.

**Parameters**

- **file_path** (*str*) – Location to save file.
- **pickle_protocol** (*int*) – The pickle data stream format.

**transform**(*self*, *X*, *y=None*)

Regularizes a dataframe and target data to an inferrable offset frequency.

A 'clean' X and y (if y was passed in) are created based on an inferrable offset frequency and matching datetime values with the original X and y are imputed into the clean X and y. Datetime values identified as misaligned are shifted into their appropriate position.

**Parameters**

- **X** (*pd.DataFrame*) – The input training data of shape [n_samples, n_features].
- **y** (*pd.Series, optional*) – The target training data of length [n_samples].

**Returns** Data with an inferrable *time_index* offset frequency.

**Return type** (pd.DataFrame, pd.Series)

**update_parameters**(*self*, *update_dict*, *reset_fit=True*)

Updates the parameter dictionary of the component.

**Parameters**

- **update_dict** (*dict*) – A dict of parameters to update.
- **reset_fit** (*bool, optional*) – If True, will set *_is_fitted* to False.

**class** evalml.pipelines.components.transformers.preprocessing.**URLFeaturizer**(*random_seed=0*, ***kwargs*)

Transformer that can automatically extract features from URL.

**Parameters** **random_seed** (*int*) – Seed for the random number generator. Defaults to 0.

**Attributes**

| | |
|---|---|
| **hyper-parameter_ranges** | {} |
| **modifies_features** | True |
| **modifies_target** | False |
| **name** | URL Featurizer |
| **training_only** | False |

**Methods**

| | |
|---|---|
| *clone* | Constructs a new component with the same parameters and random state. |
| *default_parameters* | Returns the default parameters for this component. |
| *describe* | Describe a component and its parameters. |
| *fit* | Fits component to data. |
| *fit_transform* | Fits on X and transforms X. |
| *load* | Loads component at file path. |
| *needs_fitting* | Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances. |
| *parameters* | Returns the parameters which were used to initialize the component. |
| *save* | Saves component at file path. |
| *transform* | Transforms data X. |
| *update_parameters* | Updates the parameter dictionary of the component. |

**clone**(*self*)

Constructs a new component with the same parameters and random state.

**Returns** A new instance of this component with identical parameters and random state.

**default_parameters**(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

**Returns** Default parameters for this component.

**Return type** dict

**describe**(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

**Parameters**

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description as dictionary in the format {"name": name, "parameters": parameters}

**Returns** Returns dictionary if return_dict is True, else None.

**Return type** None or dict

**fit**(*self*, *X*, *y=None*)

Fits component to data.

**Parameters**

- **X** (*pd.DataFrame*) – The input training data of shape [n_samples, n_features]
- **y** (*pd.Series, optional*) – The target training data of length [n_samples]

**Returns** self

**Raises** **MethodPropertyNotFoundError** – If component does not have a fit method or a component_obj that implements fit.

**fit_transform**(*self*, *X*, *y=None*)

Fits on X and transforms X.

**Parameters**

- **X** (*pd.DataFrame*) – Data to fit and transform.
- **y** (*pd.Series*) – Target data.

**Returns** Transformed X.

**Return type** pd.DataFrame

**Raises** **MethodPropertyNotFoundError** – If transformer does not have a transform method or a component_obj that implements transform.

**static load**(*file_path*)

Loads component at file path.

**Parameters** **file_path** (*str*) – Location to load file.

**Returns** ComponentBase object

**needs_fitting**(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need to be fit or whose fit methods do nothing.

**Returns** True.

**property parameters**(*self*)

Returns the parameters which were used to initialize the component.

**save**(*self*, *file_path*, *pickle_protocol=cloudpickle.DEFAULT_PROTOCOL*)

Saves component at file path.

**Parameters**

- **file_path** (*str*) – Location to save file.
- **pickle_protocol** (*int*) – The pickle data stream format.

**transform**(*self*, *X*, *y=None*)

Transforms data X.

**Parameters**

- **X** (*pd.DataFrame*) – Data to transform.
- **y** (*pd.Series, optional*) – Target data.

**Returns** Transformed X

**Return type** pd.DataFrame

**Raises** **MethodPropertyNotFoundError** – If transformer does not have a transform method or a component_obj that implements transform.

**update_parameters**(*self*, *update_dict*, *reset_fit=True*)

Updates the parameter dictionary of the component.

**Parameters**

- **update_dict** (*dict*) – A dict of parameters to update.
- **reset_fit** (*bool, optional*) – If True, will set *_is_fitted* to False.

### samplers

Sampler components.

### Submodules

### base_sampler

Base Sampler component. Used as the base class of all sampler components.

### Module Contents

### Classes Summary

| | |
|---|---|
| *BaseSampler* | Base Sampler component. Used as the base class of all sampler components. |

### Contents

**class** evalml.pipelines.components.transformers.samplers.base_sampler.**BaseSampler**(*parameters=None*, *component_obj=None*, *random_seed=0*, ***kwargs*)

Base Sampler component. Used as the base class of all sampler components.

**Parameters**

- **parameters** (*dict*) – Dictionary of parameters for the component. Defaults to None.
- **component_obj** (*obj*) – Third-party objects useful in component implementation. Defaults to None.
- **random_seed** (*int*) – Seed for the random number generator. Defaults to 0.

**Attributes**

| | |
|---|---|
| **modifies_features** | True |
| **modifies_target** | True |
| **training_only** | True |

**Methods**

| | |
|---|---|
| *clone* | Constructs a new component with the same parameters and random state. |
| *default_parameters* | Returns the default parameters for this component. |
| *describe* | Describe a component and its parameters. |
| *fit* | Fits the sampler to the data. |
| *fit_transform* | Fit and transform data using the sampler component. |
| *load* | Loads component at file path. |
| *name* | Returns string name of this component. |
| *needs_fitting* | Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances. |
| *parameters* | Returns the parameters which were used to initialize the component. |
| *save* | Saves component at file path. |
| *transform* | Transforms the input data by sampling the data. |
| *update_parameters* | Updates the parameter dictionary of the component. |

**clone**(*self*)

Constructs a new component with the same parameters and random state.

**Returns** A new instance of this component with identical parameters and random state.

**default_parameters**(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

**Returns** Default parameters for this component.

**Return type** dict

**describe**(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

**Parameters**

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description as dictionary in the format {"name": name, "parameters": parameters}

**Returns** Returns dictionary if return_dict is True, else None.

**Return type** None or dict

**fit**(*self*, *X*, *y*)

Fits the sampler to the data.

**Parameters**

- **X** (*pd.DataFrame*) – Input features.
- **y** (*pd.Series*) – Target.

**Returns** self

**Raises** **ValueError** – If y is None.

**fit_transform**(*self*, *X*, *y*)

Fit and transform data using the sampler component.

**Parameters**

- **X** (*pd.DataFrame*) – The input training data of shape [n_samples, n_features].
- **y** (*pd.Series, optional*) – The target training data of length [n_samples].

**Returns** Transformed data.

**Return type** (pd.DataFrame, pd.Series)

**static load**(*file_path*)

Loads component at file path.

**Parameters** **file_path** (*str*) – Location to load file.

**Returns** ComponentBase object

**property name**(*cls*)

Returns string name of this component.

**needs_fitting**(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need to be fit or whose fit methods do nothing.

**Returns** True.

**property parameters**(*self*)

Returns the parameters which were used to initialize the component.

**save**(*self*, *file_path*, *pickle_protocol=cloudpickle.DEFAULT_PROTOCOL*)

Saves component at file path.

**Parameters**

- **file_path** (*str*) – Location to save file.
- **pickle_protocol** (*int*) – The pickle data stream format.

**transform**(*self*, *X*, *y*)

Transforms the input data by sampling the data.

**Parameters**

- **X** (*pd.DataFrame*) – Training features.
- **y** (*pd.Series*) – Target.

**Returns** Transformed features and target.

**Return type** pd.DataFrame, pd.Series

**update_parameters**(*self*, *update_dict*, *reset_fit=True*)

Updates the parameter dictionary of the component.

**Parameters**

- **update_dict** (*dict*) – A dict of parameters to update.
- **reset_fit** (*bool, optional*) – If True, will set *_is_fitted* to False.

### oversampler

SMOTE Oversampler component. Will automatically select whether to use SMOTE, SMOTEN, or SMOTENC based on inputs to the component.

### Module Contents

### Classes Summary

| | |
|---|---|
| *Oversampler* | SMOTE Oversampler component. Will automatically select whether to use SMOTE, SMOTEN, or SMOTENC based on inputs to the component. |

### Contents

**class** evalml.pipelines.components.transformers.samplers.oversampler.**Oversampler**(*sampling_ratio=0.25, sampling_ratio_dict=None, k_neighbors_default=5, n_jobs=- 1, random_seed=0, **kwargs*)

SMOTE Oversampler component. Will automatically select whether to use SMOTE, SMOTEN, or SMOTENC based on inputs to the component.

**Parameters**

- **sampling_ratio** (*float*) – This is the goal ratio of the minority to majority class, with range (0, 1]. A value of 0.25 means we want a 1:4 ratio of the minority to majority class after oversampling. We will create the a sampling dictionary using this ratio, with the keys corresponding to the class and the values responding to the number of samples. Defaults to 0.25.
- **sampling_ratio_dict** (*dict*) – A dictionary specifying the desired balanced ratio for each target value. For instance, in a binary case where class 1 is the minority, we could specify: *sampling_ratio_dict={0: 0.5, 1: 1}*, which means we would undersample class 0 to have twice the number of samples as class 1 (minority:majority ratio = 0.5), and don't sample class 1. Overrides sampling_ratio if provided. Defaults to None.
- **k_neighbors_default** (*int*) – The number of nearest neighbors used to construct synthetic samples. This is the default value used, but the actual k_neighbors value might be smaller if there are less samples. Defaults to 5.
- **n_jobs** (*int*) – The number of CPU cores to use. Defaults to -1.
- **random_seed** (*int*) – The seed to use for random sampling. Defaults to 0.

**Attributes**

| | |
|---|---|
| **hyper-parameter_ranges** | None |
| **modifies_features** | True |
| **modifies_target** | True |
| **name** | Oversampler |
| **training_only** | True |

**Methods**

| | |
|---|---|
| *clone* | Constructs a new component with the same parameters and random state. |
| *default_parameters* | Returns the default parameters for this component. |
| *describe* | Describe a component and its parameters. |
| *fit* | Fits oversampler to data. |
| *fit_transform* | Fit and transform data using the sampler component. |
| *load* | Loads component at file path. |
| *needs_fitting* | Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances. |
| *parameters* | Returns the parameters which were used to initialize the component. |
| *save* | Saves component at file path. |
| *transform* | Transforms the input data by Oversampling the data. |
| *update_parameters* | Updates the parameter dictionary of the component. |

**clone**(*self*)

Constructs a new component with the same parameters and random state.

**Returns** A new instance of this component with identical parameters and random state.

**default_parameters**(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

**Returns** Default parameters for this component.

**Return type** dict

**describe**(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

**Parameters**

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description as dictionary in the format {"name": name, "parameters": parameters}

**Returns** Returns dictionary if return_dict is True, else None.

**Return type** None or dict

**fit**(*self*, *X*, *y*)

Fits oversampler to data.

**Parameters**

- **X** (*pd.DataFrame*) – The input training data of shape [n_samples, n_features].
- **y** (*pd.Series, optional*) – The target training data of length [n_samples].

**Returns** self

**fit_transform**(*self*, *X*, *y*)

Fit and transform data using the sampler component.

**Parameters**

- **X** (*pd.DataFrame*) – The input training data of shape [n_samples, n_features].
- **y** (*pd.Series, optional*) – The target training data of length [n_samples].

**Returns** Transformed data.

**Return type** (pd.DataFrame, pd.Series)

**static load**(*file_path*)

Loads component at file path.

**Parameters** **file_path** (*str*) – Location to load file.

**Returns** ComponentBase object

**needs_fitting**(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need to be fit or whose fit methods do nothing.

**Returns** True.

**property parameters**(*self*)

Returns the parameters which were used to initialize the component.

**save**(*self*, *file_path*, *pickle_protocol=cloudpickle.DEFAULT_PROTOCOL*)

Saves component at file path.

**Parameters**

- **file_path** (*str*) – Location to save file.
- **pickle_protocol** (*int*) – The pickle data stream format.

**transform**(*self*, *X*, *y=None*)

Transforms the input data by Oversampling the data.

**Parameters**

- **X** (*pd.DataFrame*) – Training features.
- **y** (*pd.Series*) – Target.

**Returns** Transformed features and target.

**Return type** pd.DataFrame, pd.Series

**update_parameters**(*self*, *update_dict*, *reset_fit=True*)

Updates the parameter dictionary of the component.

**Parameters**

- **update_dict** (*dict*) – A dict of parameters to update.
- **reset_fit** (*bool, optional*) – If True, will set *_is_fitted* to False.

### undersampler

An undersampling transformer to downsample the majority classes in the dataset.

### Module Contents

### Classes Summary

| | |
|---|---|
| *Undersampler* | Initializes an undersampling transformer to downsample the majority classes in the dataset. |

### Contents

**class** evalml.pipelines.components.transformers.samplers.undersampler.**Undersampler**(*sampling_ratio=0.25*, *sampling_ratio_dict=None*, *min_samples=100*, *min_percentage=0.1*, *random_seed=0*, ***kwargs*)

Initializes an undersampling transformer to downsample the majority classes in the dataset.

This component is only run during training and not during predict.

**Parameters**

- **sampling_ratio** (*float*) – The smallest minority:majority ratio that is accepted as 'balanced'. For instance, a 1:4 ratio would be represented as 0.25, while a 1:1 ratio is 1.0. Must be between 0 and 1, inclusive. Defaults to 0.25.
- **sampling_ratio_dict** (*dict*) – A dictionary specifying the desired balanced ratio for each target value. For instance, in a binary case where class 1 is the minority, we could specify: *sampling_ratio_dict={0: 0.5, 1: 1}*, which means we would undersample class 0 to have twice the number of samples as class 1 (minority:majority ratio = 0.5), and don't sample class 1. Overrides sampling_ratio if provided. Defaults to None.
- **min_samples** (*int*) – The minimum number of samples that we must have for any class, pre or post sampling. If a class must be downsampled, it will not be downsampled past this value. To determine severe imbalance, the minority class must occur less often than this and must have a class ratio below min_percentage. Must be greater than 0. Defaults to 100.
- **min_percentage** (*float*) – The minimum percentage of the minimum class to total dataset that we tolerate, as long as it is above min_samples. If min_percentage and min_samples

are not met, treat this as severely imbalanced, and we will not resample the data. Must be between 0 and 0.5, inclusive. Defaults to 0.1.

- **random_seed** (*int*) – The seed to use for random sampling. Defaults to 0.

**Raises**

- **ValueError** – If sampling_ratio is not in the range (0, 1].
- **ValueError** – If min_sample is not greater than 0.
- **ValueError** – If min_percentage is not between 0 and 0.5, inclusive.

**Attributes**

| | |
|---|---|
| **hyper-parameter_ranges** | {} |
| **modifies_features** | True |
| **modifies_target** | True |
| **name** | Undersampler |
| **training_only** | True |

**Methods**

| | |
|---|---|
| *clone* | Constructs a new component with the same parameters and random state. |
| *default_parameters* | Returns the default parameters for this component. |
| *describe* | Describe a component and its parameters. |
| *fit* | Fits the sampler to the data. |
| *fit_resample* | Resampling technique for this sampler. |
| *fit_transform* | Fit and transform data using the sampler component. |
| *load* | Loads component at file path. |
| *needs_fitting* | Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances. |
| *parameters* | Returns the parameters which were used to initialize the component. |
| *save* | Saves component at file path. |
| *transform* | Transforms the input data by sampling the data. |
| *update_parameters* | Updates the parameter dictionary of the component. |

**clone**(*self*)

Constructs a new component with the same parameters and random state.

**Returns** A new instance of this component with identical parameters and random state.

**default_parameters**(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

**Returns** Default parameters for this component.

**Return type** dict

**describe**(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

**Parameters**

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description as dictionary in the format {"name": name, "parameters": parameters}

**Returns** Returns dictionary if return_dict is True, else None.

**Return type** None or dict

**fit**(*self*, *X*, *y*)

Fits the sampler to the data.

**Parameters**

- **X** (*pd.DataFrame*) – Input features.
- **y** (*pd.Series*) – Target.

**Returns** self

**Raises** **ValueError** – If y is None.

**fit_resample**(*self*, *X*, *y*)

Resampling technique for this sampler.

**Parameters**

- **X** (*pd.DataFrame*) – Training data to fit and resample.
- **y** (*pd.Series*) – Training data targets to fit and resample.

**Returns** Indices to keep for training data.

**Return type** list

**fit_transform**(*self*, *X*, *y*)

Fit and transform data using the sampler component.

**Parameters**

- **X** (*pd.DataFrame*) – The input training data of shape [n_samples, n_features].
- **y** (*pd.Series, optional*) – The target training data of length [n_samples].

**Returns** Transformed data.

**Return type** (pd.DataFrame, pd.Series)

**static load**(*file_path*)

Loads component at file path.

**Parameters** **file_path** (*str*) – Location to load file.

**Returns** ComponentBase object

**needs_fitting**(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need to be fit or whose fit methods do nothing.

**Returns** True.

**property parameters**(*self*)

Returns the parameters which were used to initialize the component.

**save**(*self*, *file_path*, *pickle_protocol=cloudpickle.DEFAULT_PROTOCOL*)

Saves component at file path.

**Parameters**

- **file_path** (*str*) – Location to save file.
- **pickle_protocol** (*int*) – The pickle data stream format.

**transform**(*self*, *X*, *y=None*)

Transforms the input data by sampling the data.

**Parameters**

- **X** (*pd.DataFrame*) – Training features.
- **y** (*pd.Series*) – Target.

**Returns** Transformed features and target.

**Return type** pd.DataFrame, pd.Series

**update_parameters**(*self*, *update_dict*, *reset_fit=True*)

Updates the parameter dictionary of the component.

**Parameters**

- **update_dict** (*dict*) – A dict of parameters to update.
- **reset_fit** (*bool, optional*) – If True, will set *_is_fitted* to False.

## Package Contents

## Classes Summary

| | |
|---|---|
| *Oversampler* | SMOTE Oversampler component. Will automatically select whether to use SMOTE, SMOTEN, or SMOTENC based on inputs to the component. |
| *Undersampler* | Initializes an undersampling transformer to downsample the majority classes in the dataset. |

## Contents

**class** evalml.pipelines.components.transformers.samplers.**Oversampler**(*sampling_ratio=0.25*, *sampling_ratio_dict=None*, *k_neighbors_default=5*, *n_jobs=- 1*, *random_seed=0*, ***kwargs*)

SMOTE Oversampler component. Will automatically select whether to use SMOTE, SMOTEN, or SMOTENC based on inputs to the component.

**Parameters**

- **sampling_ratio** (*float*) – This is the goal ratio of the minority to majority class, with range (0, 1]. A value of 0.25 means we want a 1:4 ratio of the minority to majority class after oversampling. We will create the a sampling dictionary using this ratio, with the keys corresponding to the class and the values responding to the number of samples. Defaults to 0.25.
- **sampling_ratio_dict** (*dict*) – A dictionary specifying the desired balanced ratio for each target value. For instance, in a binary case where class 1 is the minority, we could specify: *sampling_ratio_dict={0: 0.5, 1: 1}*, which means we would undersample class 0 to have twice the number of samples as class 1 (minority:majority ratio = 0.5), and don't sample class 1. Overrides sampling_ratio if provided. Defaults to None.
- **k_neighbors_default** (*int*) – The number of nearest neighbors used to construct synthetic samples. This is the default value used, but the actual k_neighbors value might be smaller if there are less samples. Defaults to 5.
- **n_jobs** (*int*) – The number of CPU cores to use. Defaults to -1.
- **random_seed** (*int*) – The seed to use for random sampling. Defaults to 0.

**Attributes**

| | |
|---|---|
| **hyper-parameter_ranges** | None |
| **modifies_features** | True |
| **modifies_target** | True |
| **name** | Oversampler |
| **training_only** | True |

**Methods**

| | |
|---|---|
| *clone* | Constructs a new component with the same parameters and random state. |
| *default_parameters* | Returns the default parameters for this component. |
| *describe* | Describe a component and its parameters. |
| *fit* | Fits oversampler to data. |
| *fit_transform* | Fit and transform data using the sampler component. |
| *load* | Loads component at file path. |
| *needs_fitting* | Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances. |
| *parameters* | Returns the parameters which were used to initialize the component. |
| *save* | Saves component at file path. |
| *transform* | Transforms the input data by Oversampling the data. |
| *update_parameters* | Updates the parameter dictionary of the component. |

**clone**(*self*)

Constructs a new component with the same parameters and random state.

**Returns** A new instance of this component with identical parameters and random state.

**default_parameters**(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

**Returns** Default parameters for this component.

**Return type** dict

**describe**(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

**Parameters**

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description as dictionary in the format {"name": name, "parameters": parameters}

**Returns** Returns dictionary if return_dict is True, else None.

**Return type** None or dict

**fit**(*self*, *X*, *y*)

Fits oversampler to data.

**Parameters**

- **X** (*pd.DataFrame*) – The input training data of shape [n_samples, n_features].
- **y** (*pd.Series, optional*) – The target training data of length [n_samples].

**Returns** self

**fit_transform**(*self*, *X*, *y*)

Fit and transform data using the sampler component.

**Parameters**

- **X** (*pd.DataFrame*) – The input training data of shape [n_samples, n_features].
- **y** (*pd.Series, optional*) – The target training data of length [n_samples].

**Returns** Transformed data.

**Return type** (pd.DataFrame, pd.Series)

**static load**(*file_path*)

Loads component at file path.

**Parameters** **file_path** (*str*) – Location to load file.

**Returns** ComponentBase object

**needs_fitting**(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need to be fit or whose fit methods do nothing.

**Returns** True.

**property parameters**(*self*)

Returns the parameters which were used to initialize the component.

**save**(*self*, *file_path*, *pickle_protocol=cloudpickle.DEFAULT_PROTOCOL*)

Saves component at file path.

**Parameters**

- **file_path** (*str*) – Location to save file.
- **pickle_protocol** (*int*) – The pickle data stream format.

**transform**(*self*, *X*, *y=None*)

Transforms the input data by Oversampling the data.

**Parameters**

- **X** (*pd.DataFrame*) – Training features.
- **y** (*pd.Series*) – Target.

**Returns** Transformed features and target.

**Return type** pd.DataFrame, pd.Series

**update_parameters**(*self*, *update_dict*, *reset_fit=True*)

Updates the parameter dictionary of the component.

**Parameters**

- **update_dict** (*dict*) – A dict of parameters to update.
- **reset_fit** (*bool, optional*) – If True, will set *_is_fitted* to False.

**class** evalml.pipelines.components.transformers.samplers.**Undersampler**(*sampling_ratio=0.25*, *sampling_ratio_dict=None*, *min_samples=100*, *min_percentage=0.1*, *random_seed=0*, ***kwargs*)

Initializes an undersampling transformer to downsample the majority classes in the dataset.

This component is only run during training and not during predict.

**Parameters**

- **sampling_ratio** (*float*) – The smallest minority:majority ratio that is accepted as 'balanced'. For instance, a 1:4 ratio would be represented as 0.25, while a 1:1 ratio is 1.0. Must be between 0 and 1, inclusive. Defaults to 0.25.
- **sampling_ratio_dict** (*dict*) – A dictionary specifying the desired balanced ratio for each target value. For instance, in a binary case where class 1 is the minority, we could specify: *sampling_ratio_dict={0: 0.5, 1: 1}*, which means we would undersample class 0 to have twice the number of samples as class 1 (minority:majority ratio = 0.5), and don't sample class 1. Overrides sampling_ratio if provided. Defaults to None.
- **min_samples** (*int*) – The minimum number of samples that we must have for any class, pre or post sampling. If a class must be downsampled, it will not be downsampled past this value. To determine severe imbalance, the minority class must occur less often than this and must have a class ratio below min_percentage. Must be greater than 0. Defaults to 100.
- **min_percentage** (*float*) – The minimum percentage of the minimum class to total dataset that we tolerate, as long as it is above min_samples. If min_percentage and min_samples are not met, treat this as severely imbalanced, and we will not resample the data. Must be between 0 and 0.5, inclusive. Defaults to 0.1.

- **random_seed** (*int*) – The seed to use for random sampling. Defaults to 0.

**Raises**

- **ValueError** – If sampling_ratio is not in the range (0, 1].
- **ValueError** – If min_sample is not greater than 0.
- **ValueError** – If min_percentage is not between 0 and 0.5, inclusive.

**Attributes**

| | |
|---|---|
| **hyper-parameter_ranges** | {} |
| **modifies_features** | True |
| **modifies_target** | True |
| **name** | Undersampler |
| **training_only** | True |

**Methods**

| | |
|---|---|
| *clone* | Constructs a new component with the same parameters and random state. |
| *default_parameters* | Returns the default parameters for this component. |
| *describe* | Describe a component and its parameters. |
| *fit* | Fits the sampler to the data. |
| *fit_resample* | Resampling technique for this sampler. |
| *fit_transform* | Fit and transform data using the sampler component. |
| *load* | Loads component at file path. |
| *needs_fitting* | Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances. |
| *parameters* | Returns the parameters which were used to initialize the component. |
| *save* | Saves component at file path. |
| *transform* | Transforms the input data by sampling the data. |
| *update_parameters* | Updates the parameter dictionary of the component. |

**clone**(*self*)

Constructs a new component with the same parameters and random state.

**Returns** A new instance of this component with identical parameters and random state.

**default_parameters**(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

**Returns** Default parameters for this component.

**Return type** dict

**describe**(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

**Parameters**

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description as dictionary in the format {"name": name, "parameters": parameters}

**Returns** Returns dictionary if return_dict is True, else None.

**Return type** None or dict

**fit**(*self*, *X*, *y*)

Fits the sampler to the data.

**Parameters**

- **X** (*pd.DataFrame*) – Input features.
- **y** (*pd.Series*) – Target.

**Returns** self

**Raises** **ValueError** – If y is None.

**fit_resample**(*self*, *X*, *y*)

Resampling technique for this sampler.

**Parameters**

- **X** (*pd.DataFrame*) – Training data to fit and resample.
- **y** (*pd.Series*) – Training data targets to fit and resample.

**Returns** Indices to keep for training data.

**Return type** list

**fit_transform**(*self*, *X*, *y*)

Fit and transform data using the sampler component.

**Parameters**

- **X** (*pd.DataFrame*) – The input training data of shape [n_samples, n_features].
- **y** (*pd.Series, optional*) – The target training data of length [n_samples].

**Returns** Transformed data.

**Return type** (pd.DataFrame, pd.Series)

**static load**(*file_path*)

Loads component at file path.

**Parameters** **file_path** (*str*) – Location to load file.

**Returns** ComponentBase object

**needs_fitting**(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need to be fit or whose fit methods do nothing.

**Returns** True.

**property parameters**(*self*)

Returns the parameters which were used to initialize the component.

**save**(*self*, *file_path*, *pickle_protocol=cloudpickle.DEFAULT_PROTOCOL*)

Saves component at file path.

**Parameters**

- **file_path** (*str*) – Location to save file.
- **pickle_protocol** (*int*) – The pickle data stream format.

**transform**(*self*, *X*, *y=None*)

Transforms the input data by sampling the data.

**Parameters**

- **X** (*pd.DataFrame*) – Training features.
- **y** (*pd.Series*) – Target.

**Returns** Transformed features and target.

**Return type** pd.DataFrame, pd.Series

**update_parameters**(*self*, *update_dict*, *reset_fit=True*)

Updates the parameter dictionary of the component.

**Parameters**

- **update_dict** (*dict*) – A dict of parameters to update.
- **reset_fit** (*bool, optional*) – If True, will set *_is_fitted* to False.

### scalers

Components that scale input data.

### Submodules

### standard_scaler

A transformer that standardizes input features by removing the mean and scaling to unit variance.

### Module Contents

### Classes Summary

| | |
|---|---|
| *StandardScaler* | A transformer that standardizes input features by removing the mean and scaling to unit variance. |

## Contents

**class** evalml.pipelines.components.transformers.scalers.standard_scaler.**StandardScaler**(*random_seed=0*, ***kwargs*)

A transformer that standardizes input features by removing the mean and scaling to unit variance.

**Parameters** **random_seed** (*int*) – Seed for the random number generator. Defaults to 0.

**Attributes**

| | |
|---|---|
| **hyper-parameter_ranges** | {} |
| **modifies_features** | True |
| **modifies_target** | False |
| **name** | Standard Scaler |
| **training_only** | False |

**Methods**

| | |
|---|---|
| *clone* | Constructs a new component with the same parameters and random state. |
| *default_parameters* | Returns the default parameters for this component. |
| *describe* | Describe a component and its parameters. |
| *fit* | Fits the standard scalar on the given data. |
| *fit_transform* | Fit and transform data using the standard scaler component. |
| *load* | Loads component at file path. |
| *needs_fitting* | Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances. |
| *parameters* | Returns the parameters which were used to initialize the component. |
| *save* | Saves component at file path. |
| *transform* | Transform data using the fitted standard scaler. |
| *update_parameters* | Updates the parameter dictionary of the component. |

**clone**(*self*)

Constructs a new component with the same parameters and random state.

**Returns** A new instance of this component with identical parameters and random state.

**default_parameters**(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

**Returns** Default parameters for this component.

**Return type** dict

**describe**(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

**Parameters**

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description as dictionary in the format {"name": name, "parameters": parameters}

**Returns** Returns dictionary if return_dict is True, else None.

**Return type** None or dict

**fit**(*self*, *X*, *y=None*)

Fits the standard scalar on the given data.

**Parameters**

- **X** (*pd.DataFrame*) – The input training data of shape [n_samples, n_features].
- **y** (*pd.Series, optional*) – The target training data of length [n_samples].

**Returns** self

**fit_transform**(*self*, *X*, *y=None*)

Fit and transform data using the standard scaler component.

**Parameters**

- **X** (*pd.DataFrame*) – The input training data of shape [n_samples, n_features].
- **y** (*pd.Series, optional*) – The target training data of length [n_samples].

**Returns** Transformed data.

**Return type** pd.DataFrame

**static load**(*file_path*)

Loads component at file path.

**Parameters** **file_path** (*str*) – Location to load file.

**Returns** ComponentBase object

**needs_fitting**(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need to be fit or whose fit methods do nothing.

**Returns** True.

**property parameters**(*self*)

Returns the parameters which were used to initialize the component.

**save**(*self*, *file_path*, *pickle_protocol=cloudpickle.DEFAULT_PROTOCOL*)

Saves component at file path.

**Parameters**

- **file_path** (*str*) – Location to save file.
- **pickle_protocol** (*int*) – The pickle data stream format.

**transform**(*self*, *X*, *y=None*)

Transform data using the fitted standard scaler.

**Parameters**

- **X** (*pd.DataFrame*) – The input training data of shape [n_samples, n_features].
- **y** (*pd.Series, optional*) – The target training data of length [n_samples].

**Returns** Transformed data.

**Return type** pd.DataFrame

**update_parameters**(*self*, *update_dict*, *reset_fit=True*)

Updates the parameter dictionary of the component.

**Parameters**

- **update_dict** (*dict*) – A dict of parameters to update.
- **reset_fit** (*bool, optional*) – If True, will set *_is_fitted* to False.

## Package Contents

## Classes Summary

| | |
|---|---|
| *StandardScaler* | A transformer that standardizes input features by removing the mean and scaling to unit variance. |

## Contents

**class** evalml.pipelines.components.transformers.scalers.**StandardScaler**(*random_seed=0*, ***kwargs*)

A transformer that standardizes input features by removing the mean and scaling to unit variance.

**Parameters** **random_seed** (*int*) – Seed for the random number generator. Defaults to 0.

**Attributes**

| | |
|---|---|
| **hyperparameter_ranges** | {} |
| **modifies_features** | True |
| **modifies_target** | False |
| **name** | Standard Scaler |
| **training_only** | False |

**Methods**

| | |
|---|---|
| *clone* | Constructs a new component with the same parameters and random state. |
| *default_parameters* | Returns the default parameters for this component. |
| *describe* | Describe a component and its parameters. |
| *fit* | Fits the standard scalar on the given data. |
| *fit_transform* | Fit and transform data using the standard scaler component. |
| *load* | Loads component at file path. |
| *needs_fitting* | Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances. |
| *parameters* | Returns the parameters which were used to initialize the component. |
| *save* | Saves component at file path. |
| *transform* | Transform data using the fitted standard scaler. |
| *update_parameters* | Updates the parameter dictionary of the component. |

**clone**(*self*)

Constructs a new component with the same parameters and random state.

**Returns** A new instance of this component with identical parameters and random state.

**default_parameters**(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

**Returns** Default parameters for this component.

**Return type** dict

**describe**(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

**Parameters**

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description as dictionary in the format {"name": name, "parameters": parameters}

**Returns** Returns dictionary if return_dict is True, else None.

**Return type** None or dict

**fit**(*self*, *X*, *y=None*)

Fits the standard scalar on the given data.

**Parameters**

- **X** (*pd.DataFrame*) – The input training data of shape [n_samples, n_features].
- **y** (*pd.Series, optional*) – The target training data of length [n_samples].

**Returns** self

**fit_transform**(*self*, *X*, *y=None*)

Fit and transform data using the standard scaler component.

**Parameters**

- **X** (*pd.DataFrame*) – The input training data of shape [n_samples, n_features].
- **y** (*pd.Series, optional*) – The target training data of length [n_samples].

**Returns** Transformed data.

**Return type** pd.DataFrame

**static load**(*file_path*)

Loads component at file path.

**Parameters** **file_path** (*str*) – Location to load file.

**Returns** ComponentBase object

**needs_fitting**(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need to be fit or whose fit methods do nothing.

**Returns** True.

**property parameters**(*self*)

Returns the parameters which were used to initialize the component.

**save**(*self*, *file_path*, *pickle_protocol=cloudpickle.DEFAULT_PROTOCOL*)

Saves component at file path.

**Parameters**

- **file_path** (*str*) – Location to save file.
- **pickle_protocol** (*int*) – The pickle data stream format.

**transform**(*self*, *X*, *y=None*)

Transform data using the fitted standard scaler.

**Parameters**

- **X** (*pd.DataFrame*) – The input training data of shape [n_samples, n_features].
- **y** (*pd.Series, optional*) – The target training data of length [n_samples].

**Returns** Transformed data.

**Return type** pd.DataFrame

**update_parameters**(*self*, *update_dict*, *reset_fit=True*)

Updates the parameter dictionary of the component.

**Parameters**

- **update_dict** (*dict*) – A dict of parameters to update.
- **reset_fit** (*bool, optional*) – If True, will set *_is_fitted* to False.

### Submodules

### column_selectors

Initalizes an transformer that selects specified columns in input data.

### Module Contents

### Classes Summary

| | |
|---|---|
| *ColumnSelector* | Initalizes an transformer that selects specified columns in input data. |
| *DropColumns* | Drops specified columns in input data. |
| *SelectByType* | Selects columns by specified Woodwork logical type or semantic tag in input data. |
| *SelectColumns* | Selects specified columns in input data. |

### Contents

**class** evalml.pipelines.components.transformers.column_selectors.**ColumnSelector**(*columns=None*, *random_seed=0*, ***kwargs*)

Initalizes an transformer that selects specified columns in input data.

**Parameters**

- **columns** (*list(string)*) – List of column names, used to determine which columns to select.
- **random_seed** (*int*) – Seed for the random number generator. Defaults to 0.

**Attributes**

| | |
|---|---|
| **modifies_features** | True |
| **modifies_target** | False |
| **training_only** | False |

**Methods**

| | |
|---|---|
| *clone* | Constructs a new component with the same parameters and random state. |
| *default_parameters* | Returns the default parameters for this component. |
| *describe* | Describe a component and its parameters. |
| *fit* | Fits the transformer by checking if column names are present in the dataset. |
| *fit_transform* | Fits on X and transforms X. |
| *load* | Loads component at file path. |
| *name* | Returns string name of this component. |
| *needs_fitting* | Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances. |
| *parameters* | Returns the parameters which were used to initialize the component. |
| *save* | Saves component at file path. |
| *transform* | Transform data using fitted column selector component. |
| *update_parameters* | Updates the parameter dictionary of the component. |

**clone**(*self*)

Constructs a new component with the same parameters and random state.

**Returns** A new instance of this component with identical parameters and random state.

**default_parameters**(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

**Returns** Default parameters for this component.

**Return type** dict

**describe**(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

**Parameters**

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description as dictionary in the format {"name": name, "parameters": parameters}

**Returns** Returns dictionary if return_dict is True, else None.

**Return type** None or dict

**fit**(*self*, *X*, *y=None*)

Fits the transformer by checking if column names are present in the dataset.

**Parameters**

- **X** (*pd.DataFrame*) – Data to check.
- **y** (*pd.Series, ignored*) – Targets.

**Returns** self

**fit_transform**(*self*, *X*, *y=None*)

Fits on X and transforms X.

**Parameters**

- **X** (*pd.DataFrame*) – Data to fit and transform.
- **y** (*pd.Series*) – Target data.

**Returns** Transformed X.

**Return type** pd.DataFrame

**Raises** **MethodPropertyNotFoundError** – If transformer does not have a transform method or a component_obj that implements transform.

**static load**(*file_path*)

Loads component at file path.

**Parameters** **file_path** (*str*) – Location to load file.

**Returns** ComponentBase object

**property name**(*cls*)

Returns string name of this component.

**needs_fitting**(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need to be fit or whose fit methods do nothing.

**Returns** True.

**property parameters**(*self*)

Returns the parameters which were used to initialize the component.

**save**(*self*, *file_path*, *pickle_protocol=cloudpickle.DEFAULT_PROTOCOL*)

Saves component at file path.

**Parameters**

- **file_path** (*str*) – Location to save file.
- **pickle_protocol** (*int*) – The pickle data stream format.

**transform**(*self*, *X*, *y=None*)

Transform data using fitted column selector component.

**Parameters**

- **X** (*pd.DataFrame*) – The input training data of shape [n_samples, n_features].
- **y** (*pd.Series, optional*) – The target training data of length [n_samples].

**Returns** Transformed data.

**Return type** pd.DataFrame

**update_parameters**(*self*, *update_dict*, *reset_fit=True*)

Updates the parameter dictionary of the component.

**Parameters**

- **update_dict** (*dict*) – A dict of parameters to update.

- **reset_fit** (*bool, optional*) – If True, will set *_is_fitted* to False.

**class** evalml.pipelines.components.transformers.column_selectors.**DropColumns**(*columns=None, random_seed=0, **kwargs*)

Drops specified columns in input data.

**Parameters**

- **columns** (*list(string)*) – List of column names, used to determine which columns to drop.
- **random_seed** (*int*) – Seed for the random number generator. Defaults to 0.

**Attributes**

| | |
|---|---|
| **hyper-parameter_ranges** | {} |
| **modifies_features** | True |
| **modifies_target** | False |
| **name** | Drop Columns Transformer |
| **needs_fitting** | False |
| **training_only** | False |

**Methods**

| | |
|---|---|
| *clone* | Constructs a new component with the same parameters and random state. |
| *default_parameters* | Returns the default parameters for this component. |
| *describe* | Describe a component and its parameters. |
| *fit* | Fits the transformer by checking if column names are present in the dataset. |
| *fit_transform* | Fits on X and transforms X. |
| *load* | Loads component at file path. |
| *parameters* | Returns the parameters which were used to initialize the component. |
| *save* | Saves component at file path. |
| *transform* | Transforms data X by dropping columns. |
| *update_parameters* | Updates the parameter dictionary of the component. |

**clone**(*self*)

Constructs a new component with the same parameters and random state.

**Returns** A new instance of this component with identical parameters and random state.

**default_parameters**(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

**Returns** Default parameters for this component.

**Return type** dict

**describe**(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

**Parameters**

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description as dictionary in the format {"name": name, "parameters": parameters}

**Returns** Returns dictionary if return_dict is True, else None.

**Return type** None or dict

**fit**(*self*, *X*, *y=None*)

Fits the transformer by checking if column names are present in the dataset.

**Parameters**

- **X** (*pd.DataFrame*) – Data to check.
- **y** (*pd.Series, ignored*) – Targets.

**Returns** self

**fit_transform**(*self*, *X*, *y=None*)

Fits on X and transforms X.

**Parameters**

- **X** (*pd.DataFrame*) – Data to fit and transform.
- **y** (*pd.Series*) – Target data.

**Returns** Transformed X.

**Return type** pd.DataFrame

**Raises** **MethodPropertyNotFoundError** – If transformer does not have a transform method or a component_obj that implements transform.

**static load**(*file_path*)

Loads component at file path.

**Parameters** **file_path** (*str*) – Location to load file.

**Returns** ComponentBase object

**property parameters**(*self*)

Returns the parameters which were used to initialize the component.

**save**(*self*, *file_path*, *pickle_protocol=cloudpickle.DEFAULT_PROTOCOL*)

Saves component at file path.

**Parameters**

- **file_path** (*str*) – Location to save file.
- **pickle_protocol** (*int*) – The pickle data stream format.

**transform**(*self*, *X*, *y=None*)

Transforms data X by dropping columns.

**Parameters**

- **X** (*pd.DataFrame*) – Data to transform.

- **y** (*pd.Series, optional*) – Targets.

**Returns** Transformed X.

**Return type** pd.DataFrame

**update_parameters**(*self*, *update_dict*, *reset_fit=True*)

Updates the parameter dictionary of the component.

**Parameters**

- **update_dict** (*dict*) – A dict of parameters to update.
- **reset_fit** (*bool, optional*) – If True, will set *_is_fitted* to False.

**class** evalml.pipelines.components.transformers.column_selectors.**SelectByType**(*column_types=None*, *exclude=False*, *random_seed=0*, ***kwargs*)

Selects columns by specified Woodwork logical type or semantic tag in input data.

**Parameters**

- **column_types** (*string, ww.LogicalType, list(string), list(ww.LogicalType)*) – List of Woodwork types or tags, used to determine which columns to select or exclude.
- **exclude** (*bool*) – If true, exclude the column_types instead of including them. Defaults to False.
- **random_seed** (*int*) – Seed for the random number generator. Defaults to 0.

**Attributes**

| | |
|---|---|
| **hyper-parameter_ranges** | {} |
| **modifies_features** | True |
| **modifies_target** | False |
| **name** | Select Columns By Type Transformer |
| **needs_fitting** | False |
| **training_only** | False |

**Methods**

| | |
|---|---|
| clone | Constructs a new component with the same parameters and random state. |
| default_parameters | Returns the default parameters for this component. |
| describe | Describe a component and its parameters. |
| fit | Fits the transformer by checking if column names are present in the dataset. |
| fit_transform | Fits on X and transforms X. |
| load | Loads component at file path. |
| parameters | Returns the parameters which were used to initialize the component. |
| save | Saves component at file path. |
| transform | Transforms data X by selecting columns. |
| update_parameters | Updates the parameter dictionary of the component. |

**clone**(*self*)

Constructs a new component with the same parameters and random state.

**Returns** A new instance of this component with identical parameters and random state.

**default_parameters**(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

**Returns** Default parameters for this component.

**Return type** dict

**describe**(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

**Parameters**

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description as dictionary in the format {"name": name, "parameters": parameters}

**Returns** Returns dictionary if return_dict is True, else None.

**Return type** None or dict

**fit**(*self*, *X*, *y=None*)

Fits the transformer by checking if column names are present in the dataset.

**Parameters**

- **X** (*pd.DataFrame*) – Data to check.
- **y** (*pd.Series, ignored*) – Targets.

**Returns** self

**fit_transform**(*self*, *X*, *y=None*)

Fits on X and transforms X.

**Parameters**

- **X** (*pd.DataFrame*) – Data to fit and transform.
- **y** (*pd.Series*) – Target data.

**Returns** Transformed X.

**Return type** pd.DataFrame

**Raises** **MethodPropertyNotFoundError** – If transformer does not have a transform method or a component_obj that implements transform.

**static load**(*file_path*)

Loads component at file path.

**Parameters** **file_path** (*str*) – Location to load file.

**Returns** ComponentBase object

**property parameters**(*self*)

Returns the parameters which were used to initialize the component.

**save**(*self*, *file_path*, *pickle_protocol=cloudpickle.DEFAULT_PROTOCOL*)

Saves component at file path.

**Parameters**

- **file_path** (*str*) – Location to save file.
- **pickle_protocol** (*int*) – The pickle data stream format.

**transform**(*self*, *X*, *y=None*)

Transforms data X by selecting columns.

**Parameters**

- **X** (*pd.DataFrame*) – Data to transform.
- **y** (*pd.Series, optional*) – Targets.

**Returns** Transformed X.

**Return type** pd.DataFrame

**update_parameters**(*self*, *update_dict*, *reset_fit=True*)

Updates the parameter dictionary of the component.

**Parameters**

- **update_dict** (*dict*) – A dict of parameters to update.
- **reset_fit** (*bool, optional*) – If True, will set *_is_fitted* to False.

**class** evalml.pipelines.components.transformers.column_selectors.**SelectColumns**(*columns=None*, *random_seed=0*, ***kwargs*)

Selects specified columns in input data.

**Parameters**

- **columns** (*list(string)*) – List of column names, used to determine which columns to select. If columns are not present, they will not be selected.
- **random_seed** (*int*) – Seed for the random number generator. Defaults to 0.

**Attributes**

| | |
|---|---|
| **hyper-parameter_ranges** | {} |
| **modifies_features** | True |
| **modifies_target** | False |
| **name** | Select Columns Transformer |
| **needs_fitting** | False |
| **training_only** | False |

**Methods**

| | |
|---|---|
| *clone* | Constructs a new component with the same parameters and random state. |
| *default_parameters* | Returns the default parameters for this component. |
| *describe* | Describe a component and its parameters. |
| *fit* | Fits the transformer by checking if column names are present in the dataset. |
| *fit_transform* | Fits on X and transforms X. |
| *load* | Loads component at file path. |
| *parameters* | Returns the parameters which were used to initialize the component. |
| *save* | Saves component at file path. |
| *transform* | Transform data using fitted column selector component. |
| *update_parameters* | Updates the parameter dictionary of the component. |

**clone**(*self*)

Constructs a new component with the same parameters and random state.

**Returns** A new instance of this component with identical parameters and random state.

**default_parameters**(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

**Returns** Default parameters for this component.

**Return type** dict

**describe**(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

**Parameters**

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description as dictionary in the format {"name": name, "parameters": parameters}

**Returns** Returns dictionary if return_dict is True, else None.

**Return type** None or dict

**fit**(*self*, *X*, *y=None*)

Fits the transformer by checking if column names are present in the dataset.

**Parameters**

- **X** (*pd.DataFrame*) – Data to check.
- **y** (*pd.Series, optional*) – Targets.

**Returns** self

**fit_transform**(*self*, *X*, *y=None*)

Fits on X and transforms X.

**Parameters**

- **X** (*pd.DataFrame*) – Data to fit and transform.
- **y** (*pd.Series*) – Target data.

**Returns** Transformed X.

**Return type** pd.DataFrame

**Raises** **MethodPropertyNotFoundError** – If transformer does not have a transform method or a component_obj that implements transform.

**static load**(*file_path*)

Loads component at file path.

**Parameters** **file_path** (*str*) – Location to load file.

**Returns** ComponentBase object

**property parameters**(*self*)

Returns the parameters which were used to initialize the component.

**save**(*self*, *file_path*, *pickle_protocol=cloudpickle.DEFAULT_PROTOCOL*)

Saves component at file path.

**Parameters**

- **file_path** (*str*) – Location to save file.
- **pickle_protocol** (*int*) – The pickle data stream format.

**transform**(*self*, *X*, *y=None*)

Transform data using fitted column selector component.

**Parameters**

- **X** (*pd.DataFrame*) – The input training data of shape [n_samples, n_features].
- **y** (*pd.Series, optional*) – The target training data of length [n_samples].

**Returns** Transformed data.

**Return type** pd.DataFrame

**update_parameters**(*self*, *update_dict*, *reset_fit=True*)

Updates the parameter dictionary of the component.

**Parameters**

- **update_dict** (*dict*) – A dict of parameters to update.
- **reset_fit** (*bool, optional*) – If True, will set *_is_fitted* to False.

### transformer

A component that may or may not need fitting that transforms data. These components are used before an estimator.

### Module Contents

### Classes Summary

| | |
|---|---|
| *Transformer* | A component that may or may not need fitting that transforms data. These components are used before an estimator. |

### Contents

**class** evalml.pipelines.components.transformers.transformer.**Transformer**(*parameters=None*, *component_obj=None*, *random_seed=0*, ***kwargs*)

A component that may or may not need fitting that transforms data. These components are used before an estimator.

To implement a new Transformer, define your own class which is a subclass of Transformer, including a name and a list of acceptable ranges for any parameters to be tuned during the automl search (hyperparameters). Define an *__init__* method which sets up any necessary state and objects. Make sure your *__init__* only uses standard keyword arguments and calls *super().__init__()* with a parameters dict. You may also override the *fit*, *transform*, *fit_transform* and other methods in this class if appropriate.

To see some examples, check out the definitions of any Transformer component.

**Parameters**

- **parameters** (*dict*) – Dictionary of parameters for the component. Defaults to None.
- **component_obj** (*obj*) – Third-party objects useful in component implementation. Defaults to None.
- **random_seed** (*int*) – Seed for the random number generator. Defaults to 0.

**Attributes**

| | |
|---|---|
| **modifies_features** | True |
| **modifies_target** | False |
| **training_only** | False |

**Methods**

| | |
|---|---|
| *clone* | Constructs a new component with the same parameters and random state. |
| *default_parameters* | Returns the default parameters for this component. |
| *describe* | Describe a component and its parameters. |
| *fit* | Fits component to data. |
| *fit_transform* | Fits on X and transforms X. |
| *load* | Loads component at file path. |
| *name* | Returns string name of this component. |
| *needs_fitting* | Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances. |
| *parameters* | Returns the parameters which were used to initialize the component. |
| *save* | Saves component at file path. |
| *transform* | Transforms data X. |
| *update_parameters* | Updates the parameter dictionary of the component. |

**clone**(*self*)

Constructs a new component with the same parameters and random state.

**Returns** A new instance of this component with identical parameters and random state.

**default_parameters**(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

**Returns** Default parameters for this component.

**Return type** dict

**describe**(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

**Parameters**

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description as dictionary in the format {"name": name, "parameters": parameters}

**Returns** Returns dictionary if return_dict is True, else None.

**Return type** None or dict

**fit**(*self*, *X*, *y=None*)

Fits component to data.

**Parameters**

- **X** (*pd.DataFrame*) – The input training data of shape [n_samples, n_features]
- **y** (*pd.Series, optional*) – The target training data of length [n_samples]

**Returns** self

**Raises** **MethodPropertyNotFoundError** – If component does not have a fit method or a component_obj that implements fit.

**fit_transform**(*self*, *X*, *y=None*)

Fits on X and transforms X.

**Parameters**

- **X** (*pd.DataFrame*) – Data to fit and transform.
- **y** (*pd.Series*) – Target data.

**Returns** Transformed X.

**Return type** pd.DataFrame

**Raises** **MethodPropertyNotFoundError** – If transformer does not have a transform method or a component_obj that implements transform.

**static load**(*file_path*)

Loads component at file path.

**Parameters** **file_path** (*str*) – Location to load file.

**Returns** ComponentBase object

**property name**(*cls*)

Returns string name of this component.

**needs_fitting**(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need to be fit or whose fit methods do nothing.

**Returns** True.

**property parameters**(*self*)

Returns the parameters which were used to initialize the component.

**save**(*self*, *file_path*, *pickle_protocol=cloudpickle.DEFAULT_PROTOCOL*)

Saves component at file path.

**Parameters**

- **file_path** (*str*) – Location to save file.
- **pickle_protocol** (*int*) – The pickle data stream format.

**abstract transform**(*self*, *X*, *y=None*)

Transforms data X.

**Parameters**

- **X** (*pd.DataFrame*) – Data to transform.
- **y** (*pd.Series, optional*) – Target data.

**Returns** Transformed X

**Return type** pd.DataFrame

**Raises** **MethodPropertyNotFoundError** – If transformer does not have a transform method or a component_obj that implements transform.

**update_parameters**(*self*, *update_dict*, *reset_fit=True*)

Updates the parameter dictionary of the component.

**Parameters**

- **update_dict** (*dict*) – A dict of parameters to update.
- **reset_fit** (*bool, optional*) – If True, will set *_is_fitted* to False.

## Package Contents

## Classes Summary

| | |
|---|---|
| *DateTimeFeaturizer* | Transformer that can automatically extract features from datetime columns. |
| *DFSTransformer* | Featuretools DFS component that generates features for the input features. |
| *DropColumns* | Drops specified columns in input data. |
| *DropNaNRowsTransformer* | Transformer to drop rows with NaN values. |
| *DropNullColumns* | Transformer to drop features whose percentage of NaN values exceeds a specified threshold. |
| *DropRowsTransformer* | Transformer to drop rows specified by row indices. |
| *EmailFeaturizer* | Transformer that can automatically extract features from emails. |
| *FeatureSelector* | Selects top features based on importance weights. |
| *Imputer* | Imputes missing data according to a specified imputation strategy. |
| *LabelEncoder* | A transformer that encodes target labels using values between 0 and num_classes - 1. |
| *LinearDiscriminantAnalysis* | Reduces the number of features by using Linear Discriminant Analysis. |
| *LogTransformer* | Applies a log transformation to the target data. |
| *LSA* | Transformer to calculate the Latent Semantic Analysis Values of text input. |
| *NaturalLanguageFeaturizer* | Transformer that can automatically featurize text columns using featuretools' nlp_primitives. |
| *OneHotEncoder* | A transformer that encodes categorical features in a one-hot numeric array. |
| *OrdinalEncoder* | A transformer that encodes ordinal features as an array of ordinal integers representing the relative order of categories. |
| *Oversampler* | SMOTE Oversampler component. Will automatically select whether to use SMOTE, SMOTEN, or SMOTENC based on inputs to the component. |
| *PCA* | Reduces the number of features by using Principal Component Analysis (PCA). |
| *PerColumnImputer* | Imputes missing data according to a specified imputation strategy per column. |
| *PolynomialDecomposer* | Removes trends and seasonality from time series by fitting a polynomial and moving average to the data. |

continues on next page

Table 5 – continued from previous page

| | |
|---|---|
| *ReplaceNullableTypes* | Transformer to replace features with the new nullable dtypes with a dtype that is compatible in EvalML. |
| *RFClassifierRFESelector* | Selects relevant features using recursive feature elimination with a Random Forest Classifier. |
| *RFClassifierSelectFromModel* | Selects top features based on importance weights using a Random Forest classifier. |
| *RFRegressorRFESelector* | Selects relevant features using recursive feature elimination with a Random Forest Regressor. |
| *RFRegressorSelectFromModel* | Selects top features based on importance weights using a Random Forest regressor. |
| *SelectByType* | Selects columns by specified Woodwork logical type or semantic tag in input data. |
| *SelectColumns* | Selects specified columns in input data. |
| *SimpleImputer* | Imputes missing data according to a specified imputation strategy. Natural language columns are ignored. |
| *StandardScaler* | A transformer that standardizes input features by removing the mean and scaling to unit variance. |
| *STLDecomposer* | Removes trends and seasonality from time series using the STL algorithm. |
| *TargetEncoder* | A transformer that encodes categorical features into target encodings. |
| *TargetImputer* | Imputes missing target data according to a specified imputation strategy. |
| *TimeSeriesFeaturizer* | Transformer that delays input features and target variable for time series problems. |
| *TimeSeriesImputer* | Imputes missing data according to a specified timeseries-specific imputation strategy. |
| *TimeSeriesRegularizer* | Transformer that regularizes an inconsistently spaced datetime column. |
| *Transformer* | A component that may or may not need fitting that transforms data. These components are used before an estimator. |
| *Undersampler* | Initializes an undersampling transformer to downsample the majority classes in the dataset. |
| *URLFeaturizer* | Transformer that can automatically extract features from URL. |

## Contents

**class** evalml.pipelines.components.transformers.**DateTimeFeaturizer**(*features_to_extract=None*, *encode_as_categories=False*, *time_index=None*, *random_seed=0*, ***kwargs*)

Transformer that can automatically extract features from datetime columns.

**Parameters**

- **features_to_extract** (*list*) – List of features to extract. Valid options include "year", "month", "day_of_week", "hour". Defaults to None.
- **encode_as_categories** (*bool*) – Whether day-of-week and month features should be encoded as pandas "category" dtype. This allows OneHotEncoders to encode these features.

Defaults to False.

- **time_index** (*str*) – Name of the column containing the datetime information used to order the data. Ignored.
- **random_seed** (*int*) – Seed for the random number generator. Defaults to 0.

**Attributes**

| | |
|---|---|
| **hyper-parameter_ranges** | {} |
| **modifies_features** | True |
| **modifies_target** | False |
| **name** | DateTime Featurizer |
| **training_only** | False |

**Methods**

| | |
|---|---|
| *clone* | Constructs a new component with the same parameters and random state. |
| *default_parameters* | Returns the default parameters for this component. |
| *describe* | Describe a component and its parameters. |
| *fit* | Fit the datetime featurizer component. |
| *fit_transform* | Fits on X and transforms X. |
| *get_feature_names* | Gets the categories of each datetime feature. |
| *load* | Loads component at file path. |
| *needs_fitting* | Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances. |
| *parameters* | Returns the parameters which were used to initialize the component. |
| *save* | Saves component at file path. |
| *transform* | Transforms data X by creating new features using existing DateTime columns, and then dropping those DateTime columns. |
| *update_parameters* | Updates the parameter dictionary of the component. |

**clone**(*self*)

Constructs a new component with the same parameters and random state.

**Returns** A new instance of this component with identical parameters and random state.

**default_parameters**(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

**Returns** Default parameters for this component.

**Return type** dict

**describe**(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

**Parameters**

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description as dictionary in the format {"name": name, "parameters": parameters}

**Returns** Returns dictionary if return_dict is True, else None.

**Return type** None or dict

**fit**(*self*, *X*, *y=None*)

Fit the datetime featurizer component.

**Parameters**

- **X** (*pd.DataFrame*) – Input features.
- **y** (*pd.Series, optional*) – Target data. Ignored.

**Returns** self

**fit_transform**(*self*, *X*, *y=None*)

Fits on X and transforms X.

**Parameters**

- **X** (*pd.DataFrame*) – Data to fit and transform.
- **y** (*pd.Series*) – Target data.

**Returns** Transformed X.

**Return type** pd.DataFrame

**Raises** **MethodPropertyNotFoundError** – If transformer does not have a transform method or a component_obj that implements transform.

**get_feature_names**(*self*)

Gets the categories of each datetime feature.

**Returns**

**Dictionary, where each key-value pair is a column name and a dictionary** mapping the unique feature values to their integer encoding.

**Return type** dict

**static load**(*file_path*)

Loads component at file path.

**Parameters** **file_path** (*str*) – Location to load file.

**Returns** ComponentBase object

**needs_fitting**(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need to be fit or whose fit methods do nothing.

**Returns** True.

**property parameters**(*self*)

Returns the parameters which were used to initialize the component.

**save**(*self*, *file_path*, *pickle_protocol=cloudpickle.DEFAULT_PROTOCOL*)

Saves component at file path.

**Parameters**

- **file_path** (*str*) – Location to save file.
- **pickle_protocol** (*int*) – The pickle data stream format.

**transform**(*self*, *X*, *y=None*)

Transforms data X by creating new features using existing DateTime columns, and then dropping those DateTime columns.

**Parameters**

- **X** (*pd.DataFrame*) – Input features.
- **y** (*pd.Series, optional*) – Ignored.

**Returns** Transformed X

**Return type** pd.DataFrame

**update_parameters**(*self*, *update_dict*, *reset_fit=True*)

Updates the parameter dictionary of the component.

**Parameters**

- **update_dict** (*dict*) – A dict of parameters to update.
- **reset_fit** (*bool, optional*) – If True, will set *_is_fitted* to False.

**class** evalml.pipelines.components.transformers.**DFSTransformer**(*index='index'*, *features=None*, *random_seed=0*, ***kwargs*)

Featuretools DFS component that generates features for the input features.

**Parameters**

- **index** (*string*) – The name of the column that contains the indices. If no column with this name exists, then featuretools.EntitySet() creates a column with this name to serve as the index column. Defaults to 'index'.
- **random_seed** (*int*) – Seed for the random number generator. Defaults to 0.
- **features** (*list*) – List of features to run DFS on. Defaults to None. Features will only be computed if the columns used by the feature exist in the input and if the feature itself is not in input. If features is an empty list, no transformation will occur to inputted data.

**Attributes**

| | |
|---|---|
| **hyper-parameter_ranges** | {} |
| **modifies_features** | True |
| **modifies_target** | False |
| **name** | DFS Transformer |
| **training_only** | False |

**Methods**

| | |
|---|---|
| *clone* | Constructs a new component with the same parameters and random state. |
| *contains_pre_existing_features* | Determines whether or not features from a DFS Transformer match pipeline input features. |
| *default_parameters* | Returns the default parameters for this component. |
| *describe* | Describe a component and its parameters. |
| *fit* | Fits the DFSTransformer Transformer component. |
| *fit_transform* | Fits on X and transforms X. |
| *load* | Loads component at file path. |
| *needs_fitting* | Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances. |
| *parameters* | Returns the parameters which were used to initialize the component. |
| *save* | Saves component at file path. |
| *transform* | Computes the feature matrix for the input X using featuretools' dfs algorithm. |
| *update_parameters* | Updates the parameter dictionary of the component. |

**clone**(*self*)

Constructs a new component with the same parameters and random state.

**Returns** A new instance of this component with identical parameters and random state.

**static contains_pre_existing_features**(*dfs_features: Optional[List[featuretools.feature_base.FeatureBase]], input_feature_names: List[str], target: Optional[str] = None*)

Determines whether or not features from a DFS Transformer match pipeline input features.

**Parameters**

- **dfs_features** (*Optional[List[FeatureBase]]*) – List of features output from a DFS Transformer.
- **input_feature_names** (*List[str]*) – List of input features into the DFS Transformer.
- **target** (*Optional[str]*) – The target whose values we are trying to predict. This is used to know which column to ignore if the target column is present in the list of features in the DFS Transformer's parameters.

**default_parameters**(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

**Returns** Default parameters for this component.

**Return type** dict

**describe**(*self, print_name=False, return_dict=False*)

Describe a component and its parameters.

**Parameters**

- **print_name** (*bool, optional*) – whether to print name of component

- **return_dict** (*bool, optional*) – whether to return description as dictionary in the format {"name": name, "parameters": parameters}

**Returns** Returns dictionary if return_dict is True, else None.

**Return type** None or dict

**fit**(*self*, *X*, *y=None*)

Fits the DFSTransformer Transformer component.

**Parameters**

- **X** (*pd.DataFrame, np.array*) – The input data to transform, of shape [n_samples, n_features].
- **y** (*pd.Series*) – The target training data of length [n_samples].

**Returns** self

**fit_transform**(*self*, *X*, *y=None*)

Fits on X and transforms X.

**Parameters**

- **X** (*pd.DataFrame*) – Data to fit and transform.
- **y** (*pd.Series*) – Target data.

**Returns** Transformed X.

**Return type** pd.DataFrame

**Raises** **MethodPropertyNotFoundError** – If transformer does not have a transform method or a component_obj that implements transform.

**static load**(*file_path*)

Loads component at file path.

**Parameters** **file_path** (*str*) – Location to load file.

**Returns** ComponentBase object

**needs_fitting**(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need to be fit or whose fit methods do nothing.

**Returns** True.

**property parameters**(*self*)

Returns the parameters which were used to initialize the component.

**save**(*self*, *file_path*, *pickle_protocol=cloudpickle.DEFAULT_PROTOCOL*)

Saves component at file path.

**Parameters**

- **file_path** (*str*) – Location to save file.
- **pickle_protocol** (*int*) – The pickle data stream format.

**transform**(*self*, *X*, *y=None*)

Computes the feature matrix for the input X using featuretools' dfs algorithm.

**Parameters**

- **X** (*pd.DataFrame or np.ndarray*) – The input training data to transform. Has shape [n_samples, n_features]
- **y** (*pd.Series, optional*) – Ignored.

**Returns** Feature matrix

**Return type** pd.DataFrame

**update_parameters**(*self*, *update_dict*, *reset_fit=True*)

Updates the parameter dictionary of the component.

**Parameters**

- **update_dict** (*dict*) – A dict of parameters to update.
- **reset_fit** (*bool, optional*) – If True, will set *_is_fitted* to False.

**class** evalml.pipelines.components.transformers.**DropColumns**(*columns=None*, *random_seed=0*, ***kwargs*)

Drops specified columns in input data.

**Parameters**

- **columns** (*list(string)*) – List of column names, used to determine which columns to drop.
- **random_seed** (*int*) – Seed for the random number generator. Defaults to 0.

**Attributes**

| | |
|---|---|
| **hyper-parameter_ranges** | {} |
| **modifies_features** | True |
| **modifies_target** | False |
| **name** | Drop Columns Transformer |
| **needs_fitting** | False |
| **training_only** | False |

**Methods**

| | |
|---|---|
| clone | Constructs a new component with the same parameters and random state. |
| default_parameters | Returns the default parameters for this component. |
| describe | Describe a component and its parameters. |
| fit | Fits the transformer by checking if column names are present in the dataset. |
| fit_transform | Fits on X and transforms X. |
| load | Loads component at file path. |
| parameters | Returns the parameters which were used to initialize the component. |
| save | Saves component at file path. |
| transform | Transforms data X by dropping columns. |
| update_parameters | Updates the parameter dictionary of the component. |

**clone**(*self*)

Constructs a new component with the same parameters and random state.

**Returns** A new instance of this component with identical parameters and random state.

**default_parameters**(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

**Returns** Default parameters for this component.

**Return type** dict

**describe**(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

**Parameters**

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description as dictionary in the format {"name": name, "parameters": parameters}

**Returns** Returns dictionary if return_dict is True, else None.

**Return type** None or dict

**fit**(*self*, *X*, *y=None*)

Fits the transformer by checking if column names are present in the dataset.

**Parameters**

- **X** (*pd.DataFrame*) – Data to check.
- **y** (*pd.Series, ignored*) – Targets.

**Returns** self

**fit_transform**(*self*, *X*, *y=None*)

Fits on X and transforms X.

**Parameters**

- **X** (*pd.DataFrame*) – Data to fit and transform.
- **y** (*pd.Series*) – Target data.

**Returns** Transformed X.

**Return type** pd.DataFrame

**Raises** `MethodPropertyNotFoundError` – If transformer does not have a transform method or a component_obj that implements transform.

`static load`(*file_path*)

Loads component at file path.

**Parameters** `file_path` (*str*) – Location to load file.

**Returns** ComponentBase object

`property parameters`(*self*)

Returns the parameters which were used to initialize the component.

`save`(*self*, *file_path*, *pickle_protocol=cloudpickle.DEFAULT_PROTOCOL*)

Saves component at file path.

**Parameters**

- `file_path` (*str*) – Location to save file.
- `pickle_protocol` (*int*) – The pickle data stream format.

`transform`(*self*, *X*, *y=None*)

Transforms data X by dropping columns.

**Parameters**

- `X` (*pd.DataFrame*) – Data to transform.
- `y` (*pd.Series, optional*) – Targets.

**Returns** Transformed X.

**Return type** pd.DataFrame

`update_parameters`(*self*, *update_dict*, *reset_fit=True*)

Updates the parameter dictionary of the component.

**Parameters**

- `update_dict` (*dict*) – A dict of parameters to update.
- `reset_fit` (*bool, optional*) – If True, will set *_is_fitted* to False.

`class evalml.pipelines.components.transformers.DropNaNRowsTransformer`(*parameters=None*, *component_obj=None*, *random_seed=0*, ***kwargs*)

Transformer to drop rows with NaN values.

**Parameters** `random_seed` (*int*) – Seed for the random number generator. Is not used by this component. Defaults to 0.

**Attributes**

| | |
|---|---|
| **hyper-parameter_ranges** | {} |
| **modifies_features** | True |
| **modifies_target** | True |
| **name** | Drop NaN Rows Transformer |
| **training_only** | False |

**Methods**

| | |
|---|---|
| *clone* | Constructs a new component with the same parameters and random state. |
| *default_parameters* | Returns the default parameters for this component. |
| *describe* | Describe a component and its parameters. |
| *fit* | Fits component to data. |
| *fit_transform* | Fits on X and transforms X. |
| *load* | Loads component at file path. |
| *needs_fitting* | Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances. |
| *parameters* | Returns the parameters which were used to initialize the component. |
| *save* | Saves component at file path. |
| *transform* | Transforms data using fitted component. |
| *update_parameters* | Updates the parameter dictionary of the component. |

**clone**(*self*)

Constructs a new component with the same parameters and random state.

**Returns** A new instance of this component with identical parameters and random state.

**default_parameters**(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

**Returns** Default parameters for this component.

**Return type** dict

**describe**(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

**Parameters**

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description as dictionary in the format {"name": name, "parameters": parameters}

**Returns** Returns dictionary if return_dict is True, else None.

**Return type** None or dict

**fit**(*self*, *X*, *y=None*)

Fits component to data.

**Parameters**

- **X** (*pd.DataFrame*) – The input training data of shape [n_samples, n_features].
- **y** (*pd.Series, optional*) – The target training data of length [n_samples].

**Returns** self

**fit_transform**(*self*, *X*, *y=None*)

Fits on X and transforms X.

**Parameters**

- **X** (*pd.DataFrame*) – Data to fit and transform.
- **y** (*pd.Series*) – Target data.

**Returns** Transformed X.

**Return type** pd.DataFrame

**Raises** **MethodPropertyNotFoundError** – If transformer does not have a transform method or a component_obj that implements transform.

**static load**(*file_path*)

Loads component at file path.

**Parameters** **file_path** (*str*) – Location to load file.

**Returns** ComponentBase object

**needs_fitting**(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need to be fit or whose fit methods do nothing.

**Returns** True.

**property parameters**(*self*)

Returns the parameters which were used to initialize the component.

**save**(*self*, *file_path*, *pickle_protocol=cloudpickle.DEFAULT_PROTOCOL*)

Saves component at file path.

**Parameters**

- **file_path** (*str*) – Location to save file.
- **pickle_protocol** (*int*) – The pickle data stream format.

**transform**(*self*, *X*, *y=None*)

Transforms data using fitted component.

**Parameters**

- **X** (*pd.DataFrame*) – Features.
- **y** (*pd.Series, optional*) – Target data.

**Returns** Data with NaN rows dropped.

**Return type** (pd.DataFrame, pd.Series)

**update_parameters**(*self*, *update_dict*, *reset_fit=True*)

Updates the parameter dictionary of the component.

**Parameters**

- **update_dict** (*dict*) – A dict of parameters to update.
- **reset_fit** (*bool, optional*) – If True, will set *_is_fitted* to False.

**class** evalml.pipelines.components.transformers.**DropNullColumns**(*pct_null_threshold=1.0*, *random_seed=0*, ***kwargs*)

Transformer to drop features whose percentage of NaN values exceeds a specified threshold.

**Parameters**

- **pct_null_threshold** (*float*) – The percentage of NaN values in an input feature to drop. Must be a value between [0, 1] inclusive. If equal to 0.0, will drop columns with any null values. If equal to 1.0, will drop columns with all null values. Defaults to 0.95.
- **random_seed** (*int*) – Seed for the random number generator. Defaults to 0.

**Attributes**

| | |
|---|---|
| **hyper-parameter_ranges** | {} |
| **modifies_features** | True |
| **modifies_target** | False |
| **name** | Drop Null Columns Transformer |
| **training_only** | False |

**Methods**

| | |
|---|---|
| *clone* | Constructs a new component with the same parameters and random state. |
| *default_parameters* | Returns the default parameters for this component. |
| *describe* | Describe a component and its parameters. |
| *fit* | Fits component to data. |
| *fit_transform* | Fits on X and transforms X. |
| *load* | Loads component at file path. |
| *needs_fitting* | Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances. |
| *parameters* | Returns the parameters which were used to initialize the component. |
| *save* | Saves component at file path. |
| *transform* | Transforms data X by dropping columns that exceed the threshold of null values. |
| *update_parameters* | Updates the parameter dictionary of the component. |

**clone**(*self*)

Constructs a new component with the same parameters and random state.

**Returns** A new instance of this component with identical parameters and random state.

`default_parameters`(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

**Returns** Default parameters for this component.

**Return type** dict

`describe`(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

**Parameters**

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description as dictionary in the format {"name": name, "parameters": parameters}

**Returns** Returns dictionary if return_dict is True, else None.

**Return type** None or dict

`fit`(*self*, *X*, *y=None*)

Fits component to data.

**Parameters**

- **X** (*pd.DataFrame*) – The input training data of shape [n_samples, n_features].
- **y** (*pd.Series, optional*) – The target training data of length [n_samples].

**Returns** self

`fit_transform`(*self*, *X*, *y=None*)

Fits on X and transforms X.

**Parameters**

- **X** (*pd.DataFrame*) – Data to fit and transform.
- **y** (*pd.Series*) – Target data.

**Returns** Transformed X.

**Return type** pd.DataFrame

**Raises** **MethodPropertyNotFoundError** – If transformer does not have a transform method or a component_obj that implements transform.

`static load`(*file_path*)

Loads component at file path.

**Parameters** **file_path** (*str*) – Location to load file.

**Returns** ComponentBase object

`needs_fitting`(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need to be fit or whose fit methods do nothing.

**Returns** True.

**property parameters**(*self*)

Returns the parameters which were used to initialize the component.

**save**(*self*, *file_path*, *pickle_protocol=cloudpickle.DEFAULT_PROTOCOL*)

Saves component at file path.

**Parameters**

- **file_path** (*str*) – Location to save file.
- **pickle_protocol** (*int*) – The pickle data stream format.

**transform**(*self*, *X*, *y=None*)

Transforms data X by dropping columns that exceed the threshold of null values.

**Parameters**

- **X** (*pd.DataFrame*) – Data to transform
- **y** (*pd.Series, optional*) – Ignored.

**Returns** Transformed X

**Return type** pd.DataFrame

**update_parameters**(*self*, *update_dict*, *reset_fit=True*)

Updates the parameter dictionary of the component.

**Parameters**

- **update_dict** (*dict*) – A dict of parameters to update.
- **reset_fit** (*bool, optional*) – If True, will set *_is_fitted* to False.

**class** evalml.pipelines.components.transformers.**DropRowsTransformer**(*indices_to_drop=None*, *random_seed=0*)

Transformer to drop rows specified by row indices.

**Parameters**

- **indices_to_drop** (*list*) – List of indices to drop in the input data. Defaults to None.
- **random_seed** (*int*) – Seed for the random number generator. Is not used by this component. Defaults to 0.

**Attributes**

| | |
|---|---|
| **hyper-parameter_ranges** | {} |
| **modifies_features** | True |
| **modifies_target** | True |
| **name** | Drop Rows Transformer |
| **training_only** | True |

**Methods**

| | |
|---|---|
| *clone* | Constructs a new component with the same parameters and random state. |
| *default_parameters* | Returns the default parameters for this component. |
| *describe* | Describe a component and its parameters. |
| *fit* | Fits component to data. |
| *fit_transform* | Fits on X and transforms X. |
| *load* | Loads component at file path. |
| *needs_fitting* | Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances. |
| *parameters* | Returns the parameters which were used to initialize the component. |
| *save* | Saves component at file path. |
| *transform* | Transforms data using fitted component. |
| *update_parameters* | Updates the parameter dictionary of the component. |

**clone**(*self*)

Constructs a new component with the same parameters and random state.

**Returns** A new instance of this component with identical parameters and random state.

**default_parameters**(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

**Returns** Default parameters for this component.

**Return type** dict

**describe**(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

**Parameters**

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description as dictionary in the format {"name": name, "parameters": parameters}

**Returns** Returns dictionary if return_dict is True, else None.

**Return type** None or dict

**fit**(*self*, *X*, *y=None*)

Fits component to data.

**Parameters**

- **X** (*pd.DataFrame*) – The input training data of shape [n_samples, n_features].
- **y** (*pd.Series, optional*) – The target training data of length [n_samples].

**Returns** self

**Raises** **ValueError** – If indices to drop do not exist in input features or target.

**fit_transform**(*self*, *X*, *y=None*)

Fits on X and transforms X.

**Parameters**

- **X** (*pd.DataFrame*) – Data to fit and transform.
- **y** (*pd.Series*) – Target data.

**Returns** Transformed X.

**Return type** pd.DataFrame

**Raises** **MethodPropertyNotFoundError** – If transformer does not have a transform method or a component_obj that implements transform.

**static load**(*file_path*)

Loads component at file path.

**Parameters** **file_path** (*str*) – Location to load file.

**Returns** ComponentBase object

**needs_fitting**(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need to be fit or whose fit methods do nothing.

**Returns** True.

**property parameters**(*self*)

Returns the parameters which were used to initialize the component.

**save**(*self*, *file_path*, *pickle_protocol=cloudpickle.DEFAULT_PROTOCOL*)

Saves component at file path.

**Parameters**

- **file_path** (*str*) – Location to save file.
- **pickle_protocol** (*int*) – The pickle data stream format.

**transform**(*self*, *X*, *y=None*)

Transforms data using fitted component.

**Parameters**

- **X** (*pd.DataFrame*) – Features.
- **y** (*pd.Series, optional*) – Target data.

**Returns** Data with row indices dropped.

**Return type** (pd.DataFrame, pd.Series)

**update_parameters**(*self*, *update_dict*, *reset_fit=True*)

Updates the parameter dictionary of the component.

**Parameters**

- **update_dict** (*dict*) – A dict of parameters to update.
- **reset_fit** (*bool, optional*) – If True, will set *_is_fitted* to False.

**class** evalml.pipelines.components.transformers.**EmailFeaturizer**(*random_seed=0*, ***kwargs*)

Transformer that can automatically extract features from emails.

**Parameters** **random_seed** (*int*) – Seed for the random number generator. Defaults to 0.

**Attributes**

| | |
|---|---|
| **hyper-parameter_ranges** | {} |
| **modifies_features** | True |
| **modifies_target** | False |
| **name** | Email Featurizer |
| **training_only** | False |

**Methods**

| | |
|---|---|
| *clone* | Constructs a new component with the same parameters and random state. |
| *default_parameters* | Returns the default parameters for this component. |
| *describe* | Describe a component and its parameters. |
| *fit* | Fits component to data. |
| *fit_transform* | Fits on X and transforms X. |
| *load* | Loads component at file path. |
| *needs_fitting* | Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances. |
| *parameters* | Returns the parameters which were used to initialize the component. |
| *save* | Saves component at file path. |
| *transform* | Transforms data X. |
| *update_parameters* | Updates the parameter dictionary of the component. |

**clone**(*self*)

Constructs a new component with the same parameters and random state.

**Returns** A new instance of this component with identical parameters and random state.

**default_parameters**(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

**Returns** Default parameters for this component.

**Return type** dict

**describe**(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

**Parameters**

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description as dictionary in the format {"name": name, "parameters": parameters}

**Returns** Returns dictionary if return_dict is True, else None.

**Return type** None or dict

`fit`(*self*, *X*, *y=None*)

Fits component to data.

**Parameters**

- **X** (*pd.DataFrame*) – The input training data of shape [n_samples, n_features]
- **y** (*pd.Series, optional*) – The target training data of length [n_samples]

**Returns** self

**Raises** **MethodPropertyNotFoundError** – If component does not have a fit method or a component_obj that implements fit.

`fit_transform`(*self*, *X*, *y=None*)

Fits on X and transforms X.

**Parameters**

- **X** (*pd.DataFrame*) – Data to fit and transform.
- **y** (*pd.Series*) – Target data.

**Returns** Transformed X.

**Return type** pd.DataFrame

**Raises** **MethodPropertyNotFoundError** – If transformer does not have a transform method or a component_obj that implements transform.

`static load`(*file_path*)

Loads component at file path.

**Parameters** **file_path** (*str*) – Location to load file.

**Returns** ComponentBase object

`needs_fitting`(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need to be fit or whose fit methods do nothing.

**Returns** True.

`property parameters`(*self*)

Returns the parameters which were used to initialize the component.

`save`(*self*, *file_path*, *pickle_protocol=cloudpickle.DEFAULT_PROTOCOL*)

Saves component at file path.

**Parameters**

- **file_path** (*str*) – Location to save file.
- **pickle_protocol** (*int*) – The pickle data stream format.

`transform`(*self*, *X*, *y=None*)

Transforms data X.

**Parameters**

- **X** (*pd.DataFrame*) – Data to transform.

- **y** (*pd.Series, optional*) – Target data.

**Returns** Transformed X

**Return type** pd.DataFrame

**Raises** **MethodPropertyNotFoundError** – If transformer does not have a transform method or a component_obj that implements transform.

**update_parameters**(*self*, *update_dict*, *reset_fit=True*)

Updates the parameter dictionary of the component.

**Parameters**

- **update_dict** (*dict*) – A dict of parameters to update.
- **reset_fit** (*bool, optional*) – If True, will set *_is_fitted* to False.

**class** evalml.pipelines.components.transformers.**FeatureSelector**(*parameters=None*, *component_obj=None*, *random_seed=0*, ***kwargs*)

Selects top features based on importance weights.

**Parameters**

- **parameters** (*dict*) – Dictionary of parameters for the component. Defaults to None.
- **component_obj** (*obj*) – Third-party objects useful in component implementation. Defaults to None.
- **random_seed** (*int*) – Seed for the random number generator. Defaults to 0.

**Attributes**

| | |
|---|---|
| **modifies_features** | True |
| **modifies_target** | False |
| **training_only** | False |

**Methods**

| | |
|---|---|
| *clone* | Constructs a new component with the same parameters and random state. |
| *default_parameters* | Returns the default parameters for this component. |
| *describe* | Describe a component and its parameters. |
| *fit* | Fits component to data. |
| *fit_transform* | Fit and transform data using the feature selector. |
| *get_names* | Get names of selected features. |
| *load* | Loads component at file path. |
| *name* | Returns string name of this component. |
| *needs_fitting* | Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances. |
| *parameters* | Returns the parameters which were used to initialize the component. |
| *save* | Saves component at file path. |
| *transform* | Transforms input data by selecting features. If the component_obj does not have a transform method, will raise an MethodPropertyNotFoundError exception. |
| *update_parameters* | Updates the parameter dictionary of the component. |

**clone**(*self*)

Constructs a new component with the same parameters and random state.

**Returns** A new instance of this component with identical parameters and random state.

**default_parameters**(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

**Returns** Default parameters for this component.

**Return type** dict

**describe**(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

**Parameters**

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description as dictionary in the format {"name": name, "parameters": parameters}

**Returns** Returns dictionary if return_dict is True, else None.

**Return type** None or dict

**fit**(*self*, *X*, *y=None*)

Fits component to data.

**Parameters**

- **X** (*pd.DataFrame*) – The input training data of shape [n_samples, n_features]
- **y** (*pd.Series, optional*) – The target training data of length [n_samples]

**Returns** self

**Raises** `MethodPropertyNotFoundError` – If component does not have a fit method or a component_obj that implements fit.

`fit_transform`(*self*, *X*, *y=None*)

Fit and transform data using the feature selector.

**Parameters**

- `X` (*pd.DataFrame*) – The input training data of shape [n_samples, n_features].
- `y` (*pd.Series, optional*) – The target training data of length [n_samples].

**Returns** Transformed data.

**Return type** pd.DataFrame

`get_names`(*self*)

Get names of selected features.

**Returns** List of the names of features selected.

**Return type** list[str]

`static load`(*file_path*)

Loads component at file path.

**Parameters** `file_path` (*str*) – Location to load file.

**Returns** ComponentBase object

`property name`(*cls*)

Returns string name of this component.

`needs_fitting`(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need to be fit or whose fit methods do nothing.

**Returns** True.

`property parameters`(*self*)

Returns the parameters which were used to initialize the component.

`save`(*self*, *file_path*, *pickle_protocol=cloudpickle.DEFAULT_PROTOCOL*)

Saves component at file path.

**Parameters**

- `file_path` (*str*) – Location to save file.
- `pickle_protocol` (*int*) – The pickle data stream format.

`transform`(*self*, *X*, *y=None*)

Transforms input data by selecting features. If the component_obj does not have a transform method, will raise an MethodPropertyNotFoundError exception.

**Parameters**

- `X` (*pd.DataFrame*) – Data to transform.
- `y` (*pd.Series, optional*) – Target data. Ignored.

**Returns** Transformed X

**Return type** pd.DataFrame

**Raises** **MethodPropertyNotFoundError** – If feature selector does not have a transform method or a component_obj that implements transform

**update_parameters**(*self*, *update_dict*, *reset_fit=True*)

Updates the parameter dictionary of the component.

**Parameters**

- **update_dict** (*dict*) – A dict of parameters to update.
- **reset_fit** (*bool, optional*) – If True, will set *_is_fitted* to False.

**class** evalml.pipelines.components.transformers.**Imputer**(*categorical_impute_strategy='most_frequent'*, *categorical_fill_value=None*, *numeric_impute_strategy='mean'*, *numeric_fill_value=None*, *boolean_impute_strategy='most_frequent'*, *boolean_fill_value=None*, *random_seed=0*, ***kwargs*)

Imputes missing data according to a specified imputation strategy.

**Parameters**

- **categorical_impute_strategy** (*string*) – Impute strategy to use for string, object, boolean, categorical dtypes. Valid values include "most_frequent" and "constant".
- **numeric_impute_strategy** (*string*) – Impute strategy to use for numeric columns. Valid values include "mean", "median", "most_frequent", and "constant".
- **boolean_impute_strategy** (*string*) – Impute strategy to use for boolean columns. Valid values include "most_frequent" and "constant".
- **categorical_fill_value** (*string*) – When categorical_impute_strategy == "constant", fill_value is used to replace missing data. The default value of None will fill with the string "missing_value".
- **numeric_fill_value** (*int, float*) – When numeric_impute_strategy == "constant", fill_value is used to replace missing data. The default value of None will fill with 0.
- **boolean_fill_value** (*bool*) – When boolean_impute_strategy == "constant", fill_value is used to replace missing data. The default value of None will fill with True.
- **random_seed** (*int*) – Seed for the random number generator. Defaults to 0.

**Attributes**

| | |
|---|---|
| **hyper-parameter_ranges** | { "categorical_impute_strategy": ["most_frequent"], "numeric_impute_strategy": ["mean", "median", "most_frequent", "knn"], "boolean_impute_strategy": ["most_frequent"]} |
| **modifies_features** | True |
| **modifies_target** | False |
| **name** | Imputer |
| **training_only** | False |

**Methods**

| | |
|---|---|
| *clone* | Constructs a new component with the same parameters and random state. |
| *default_parameters* | Returns the default parameters for this component. |
| *describe* | Describe a component and its parameters. |
| *fit* | Fits imputer to data. 'None' values are converted to np.nan before imputation and are treated as the same. |
| *fit_transform* | Fits on X and transforms X. |
| *load* | Loads component at file path. |
| *needs_fitting* | Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances. |
| *parameters* | Returns the parameters which were used to initialize the component. |
| *save* | Saves component at file path. |
| *transform* | Transforms data X by imputing missing values. |
| *update_parameters* | Updates the parameter dictionary of the component. |

**clone**(*self*)

Constructs a new component with the same parameters and random state.

**Returns** A new instance of this component with identical parameters and random state.

**default_parameters**(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

**Returns** Default parameters for this component.

**Return type** dict

**describe**(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

**Parameters**

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description as dictionary in the format {"name": name, "parameters": parameters}

**Returns** Returns dictionary if return_dict is True, else None.

**Return type** None or dict

**fit**(*self*, *X*, *y=None*)

Fits imputer to data. 'None' values are converted to np.nan before imputation and are treated as the same.

**Parameters**

- **X** (*pd.DataFrame, np.ndarray*) – The input training data of shape [n_samples, n_features]
- **y** (*pd.Series, optional*) – The target training data of length [n_samples]

**Returns** self

**fit_transform**(*self*, *X*, *y=None*)

Fits on X and transforms X.

**Parameters**

- **X** (*pd.DataFrame*) – Data to fit and transform.
- **y** (*pd.Series*) – Target data.

**Returns** Transformed X.

**Return type** pd.DataFrame

**Raises** **MethodPropertyNotFoundError** – If transformer does not have a transform method or a component_obj that implements transform.

**static load**(*file_path*)

Loads component at file path.

**Parameters** **file_path** (*str*) – Location to load file.

**Returns** ComponentBase object

**needs_fitting**(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need to be fit or whose fit methods do nothing.

**Returns** True.

**property parameters**(*self*)

Returns the parameters which were used to initialize the component.

**save**(*self*, *file_path*, *pickle_protocol=cloudpickle.DEFAULT_PROTOCOL*)

Saves component at file path.

**Parameters**

- **file_path** (*str*) – Location to save file.
- **pickle_protocol** (*int*) – The pickle data stream format.

**transform**(*self*, *X*, *y=None*)

Transforms data X by imputing missing values.

**Parameters**

- **X** (*pd.DataFrame*) – Data to transform
- **y** (*pd.Series, optional*) – Ignored.

**Returns** Transformed X

**Return type** pd.DataFrame

**update_parameters**(*self*, *update_dict*, *reset_fit=True*)

Updates the parameter dictionary of the component.

**Parameters**

- **update_dict** (*dict*) – A dict of parameters to update.
- **reset_fit** (*bool, optional*) – If True, will set *_is_fitted* to False.

**class** evalml.pipelines.components.transformers.**LabelEncoder**(*positive_label=None*, *random_seed=0*, ***kwargs*)

A transformer that encodes target labels using values between 0 and num_classes - 1.

**Parameters**

- **positive_label** (*int, str*) – The label for the class that should be treated as positive (1) for binary classification problems. Ignored for multiclass problems. Defaults to None.
- **random_seed** (*int*) – Seed for the random number generator. Defaults to 0. Ignored.

**Attributes**

| | |
|---|---|
| **hyper-parameter_ranges** | {} |
| **modifies_features** | False |
| **modifies_target** | True |
| **name** | Label Encoder |
| **training_only** | False |

**Methods**

| | |
|---|---|
| *clone* | Constructs a new component with the same parameters and random state. |
| *default_parameters* | Returns the default parameters for this component. |
| *describe* | Describe a component and its parameters. |
| *fit* | Fits the label encoder. |
| *fit_transform* | Fit and transform data using the label encoder. |
| *inverse_transform* | Decodes the target data. |
| *load* | Loads component at file path. |
| *needs_fitting* | Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances. |
| *parameters* | Returns the parameters which were used to initialize the component. |
| *save* | Saves component at file path. |
| *transform* | Transform the target using the fitted label encoder. |
| *update_parameters* | Updates the parameter dictionary of the component. |

**clone**(*self*)

Constructs a new component with the same parameters and random state.

**Returns** A new instance of this component with identical parameters and random state.

**default_parameters**(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

**Returns** Default parameters for this component.

**Return type** dict

**describe**(*self, print_name=False, return_dict=False*)

Describe a component and its parameters.

**Parameters**

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description as dictionary in the format {"name": name, "parameters": parameters}

**Returns** Returns dictionary if return_dict is True, else None.

**Return type** None or dict

**fit**(*self*, *X*, *y*)

Fits the label encoder.

**Parameters**

- **X** (*pd.DataFrame*) – The input training data of shape [n_samples, n_features]. Ignored.
- **y** (*pd.Series*) – The target training data of length [n_samples].

**Returns** self

**Raises** **ValueError** – If input *y* is None.

**fit_transform**(*self*, *X*, *y*)

Fit and transform data using the label encoder.

**Parameters**

- **X** (*pd.DataFrame*) – The input training data of shape [n_samples, n_features].
- **y** (*pd.Series*) – The target training data of length [n_samples].

**Returns** The original features and an encoded version of the target.

**Return type** pd.DataFrame, pd.Series

**inverse_transform**(*self*, *y*)

Decodes the target data.

**Parameters** **y** (*pd.Series*) – Target data.

**Returns** The decoded version of the target.

**Return type** pd.Series

**Raises** **ValueError** – If input *y* is None.

**static load**(*file_path*)

Loads component at file path.

**Parameters** **file_path** (*str*) – Location to load file.

**Returns** ComponentBase object

**needs_fitting**(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need to be fit or whose fit methods do nothing.

**Returns** True.

**property parameters**(*self*)

Returns the parameters which were used to initialize the component.

**save**(*self*, *file_path*, *pickle_protocol=cloudpickle.DEFAULT_PROTOCOL*)

Saves component at file path.

**Parameters**

- **file_path** (*str*) – Location to save file.
- **pickle_protocol** (*int*) – The pickle data stream format.

**transform**(*self*, *X*, *y=None*)

Transform the target using the fitted label encoder.

**Parameters**

- **X** (*pd.DataFrame*) – The input training data of shape [n_samples, n_features]. Ignored.
- **y** (*pd.Series*) – The target training data of length [n_samples].

**Returns** The original features and an encoded version of the target.

**Return type** pd.DataFrame, pd.Series

**Raises** **ValueError** – If input *y* is None.

**update_parameters**(*self*, *update_dict*, *reset_fit=True*)

Updates the parameter dictionary of the component.

**Parameters**

- **update_dict** (*dict*) – A dict of parameters to update.
- **reset_fit** (*bool, optional*) – If True, will set *_is_fitted* to False.

**class** evalml.pipelines.components.transformers.**LinearDiscriminantAnalysis**(*n_components=None*, *random_seed=0*, ***kwargs*)

Reduces the number of features by using Linear Discriminant Analysis.

**Parameters**

- **n_components** (*int*) – The number of features to maintain after computation. Defaults to None.
- **random_seed** (*int*) – Seed for the random number generator. Defaults to 0.

**Attributes**

| | |
|---|---|
| **hyper-parameter_ranges** | {} |
| **modifies_features** | True |
| **modifies_target** | False |
| **name** | Linear Discriminant Analysis Transformer |
| **training_only** | False |

**Methods**

| | |
|---|---|
| *clone* | Constructs a new component with the same parameters and random state. |
| *default_parameters* | Returns the default parameters for this component. |
| *describe* | Describe a component and its parameters. |
| *fit* | Fits the LDA component. |
| *fit_transform* | Fit and transform data using the LDA component. |
| *load* | Loads component at file path. |
| *needs_fitting* | Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances. |
| *parameters* | Returns the parameters which were used to initialize the component. |
| *save* | Saves component at file path. |
| *transform* | Transform data using the fitted LDA component. |
| *update_parameters* | Updates the parameter dictionary of the component. |

**clone**(*self*)

Constructs a new component with the same parameters and random state.

**Returns** A new instance of this component with identical parameters and random state.

**default_parameters**(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

**Returns** Default parameters for this component.

**Return type** dict

**describe**(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

**Parameters**

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description as dictionary in the format {"name": name, "parameters": parameters}

**Returns** Returns dictionary if return_dict is True, else None.

**Return type** None or dict

**fit**(*self*, *X*, *y*)

Fits the LDA component.

**Parameters**

- **X** (*pd.DataFrame*) – The input training data of shape [n_samples, n_features].
- **y** (*pd.Series, optional*) – The target training data of length [n_samples].

**Returns** self

**Raises** **ValueError** – If input data is not all numeric.

**fit_transform**(*self*, *X*, *y=None*)

Fit and transform data using the LDA component.

**Parameters**

- **X** (*pd.DataFrame*) – The input training data of shape [n_samples, n_features].
- **y** (*pd.Series, optional*) – The target training data of length [n_samples].

**Returns** Transformed data.

**Return type** pd.DataFrame

**Raises** **ValueError** – If input data is not all numeric.

**static load**(*file_path*)

Loads component at file path.

**Parameters** **file_path** (*str*) – Location to load file.

**Returns** ComponentBase object

**needs_fitting**(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need to be fit or whose fit methods do nothing.

**Returns** True.

**property parameters**(*self*)

Returns the parameters which were used to initialize the component.

**save**(*self*, *file_path*, *pickle_protocol=cloudpickle.DEFAULT_PROTOCOL*)

Saves component at file path.

**Parameters**

- **file_path** (*str*) – Location to save file.
- **pickle_protocol** (*int*) – The pickle data stream format.

**transform**(*self*, *X*, *y=None*)

Transform data using the fitted LDA component.

**Parameters**

- **X** (*pd.DataFrame*) – The input training data of shape [n_samples, n_features].
- **y** (*pd.Series, optional*) – The target training data of length [n_samples].

**Returns** Transformed data.

**Return type** pd.DataFrame

**Raises** **ValueError** – If input data is not all numeric.

**update_parameters**(*self*, *update_dict*, *reset_fit=True*)

Updates the parameter dictionary of the component.

**Parameters**

- **update_dict** (*dict*) – A dict of parameters to update.
- **reset_fit** (*bool, optional*) – If True, will set *_is_fitted* to False.

**class** evalml.pipelines.components.transformers.**LogTransformer**(*random_seed=0*)

Applies a log transformation to the target data.

**Attributes**

| | |
|---|---|
| **hyper-parameter_ranges** | {} |
| **modifies_features** | False |
| **modifies_target** | True |
| **name** | Log Transformer |
| **training_only** | False |

**Methods**

| | |
|---|---|
| *clone* | Constructs a new component with the same parameters and random state. |
| *default_parameters* | Returns the default parameters for this component. |
| *describe* | Describe a component and its parameters. |
| *fit* | Fits the LogTransformer. |
| *fit_transform* | Log transforms the target variable. |
| *inverse_transform* | Apply exponential to target data. |
| *load* | Loads component at file path. |
| *needs_fitting* | Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances. |
| *parameters* | Returns the parameters which were used to initialize the component. |
| *save* | Saves component at file path. |
| *transform* | Log transforms the target variable. |
| *update_parameters* | Updates the parameter dictionary of the component. |

**clone**(*self*)

Constructs a new component with the same parameters and random state.

**Returns** A new instance of this component with identical parameters and random state.

**default_parameters**(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

**Returns** Default parameters for this component.

**Return type** dict

**describe**(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

**Parameters**

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description as dictionary in the format {"name": name, "parameters": parameters}

**Returns** Returns dictionary if return_dict is True, else None.

**Return type** None or dict

**fit**(*self*, *X*, *y=None*)

Fits the LogTransformer.

**Parameters**

- **X** (*pd.DataFrame or np.ndarray*) – Ignored.
- **y** (*pd.Series, optional*) – Ignored.

**Returns** self

**fit_transform**(*self*, *X*, *y=None*)

Log transforms the target variable.

**Parameters**

- **X** (*pd.DataFrame, optional*) – Ignored.
- **y** (*pd.Series*) – Target variable to log transform.

**Returns**

**The input features are returned without modification. The target** variable y is log transformed.

**Return type** tuple of pd.DataFrame, pd.Series

**inverse_transform**(*self*, *y*)

Apply exponential to target data.

**Parameters** **y** (*pd.Series*) – Target variable.

**Returns** Target with exponential applied.

**Return type** pd.Series

**static load**(*file_path*)

Loads component at file path.

**Parameters** **file_path** (*str*) – Location to load file.

**Returns** ComponentBase object

**needs_fitting**(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need to be fit or whose fit methods do nothing.

**Returns** True.

**property parameters**(*self*)

Returns the parameters which were used to initialize the component.

**save**(*self*, *file_path*, *pickle_protocol=cloudpickle.DEFAULT_PROTOCOL*)

Saves component at file path.

**Parameters**

- **file_path** (*str*) – Location to save file.
- **pickle_protocol** (*int*) – The pickle data stream format.

**transform**(*self*, *X*, *y=None*)

Log transforms the target variable.

**Parameters**

- **X** (*pd.DataFrame, optional*) – Ignored.
- **y** (*pd.Series*) – Target data to log transform.

**Returns**

**The input features are returned without modification. The target** variable y is log transformed.

**Return type** tuple of pd.DataFrame, pd.Series

**update_parameters**(*self*, *update_dict*, *reset_fit=True*)

Updates the parameter dictionary of the component.

**Parameters**

- **update_dict** (*dict*) – A dict of parameters to update.
- **reset_fit** (*bool, optional*) – If True, will set *_is_fitted* to False.

**class** evalml.pipelines.components.transformers.**LSA**(*random_seed=0*, ***kwargs*)

Transformer to calculate the Latent Semantic Analysis Values of text input.

**Parameters** **random_seed** (*int*) – Seed for the random number generator. Defaults to 0.

**Attributes**

| | |
|---|---|
| **hyper-parameter_ranges** | {} |
| **modifies_features** | True |
| **modifies_target** | False |
| **name** | LSA Transformer |
| **training_only** | False |

**Methods**

| | |
|---|---|
| *clone* | Constructs a new component with the same parameters and random state. |
| *default_parameters* | Returns the default parameters for this component. |
| *describe* | Describe a component and its parameters. |
| *fit* | Fits the input data. |
| *fit_transform* | Fits on X and transforms X. |
| *load* | Loads component at file path. |
| *needs_fitting* | Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances. |
| *parameters* | Returns the parameters which were used to initialize the component. |
| *save* | Saves component at file path. |
| *transform* | Transforms data X by applying the LSA pipeline. |
| *update_parameters* | Updates the parameter dictionary of the component. |

**clone**(*self*)

Constructs a new component with the same parameters and random state.

**Returns** A new instance of this component with identical parameters and random state.

**default_parameters**(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

**Returns** Default parameters for this component.

**Return type** dict

**describe**(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

**Parameters**

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description as dictionary in the format {"name": name, "parameters": parameters}

**Returns** Returns dictionary if return_dict is True, else None.

**Return type** None or dict

**fit**(*self*, *X*, *y=None*)

Fits the input data.

**Parameters**

- **X** (*pd.DataFrame*) – The data to transform.
- **y** (*pd.Series, optional*) – Ignored.

**Returns** self

**fit_transform**(*self*, *X*, *y=None*)

Fits on X and transforms X.

**Parameters**

- **X** (*pd.DataFrame*) – Data to fit and transform.

- **y** (*pd.Series*) – Target data.

**Returns** Transformed X.

**Return type** pd.DataFrame

**Raises** **MethodPropertyNotFoundError** – If transformer does not have a transform method or a component_obj that implements transform.

**static load**(*file_path*)

Loads component at file path.

**Parameters** **file_path** (*str*) – Location to load file.

**Returns** ComponentBase object

**needs_fitting**(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need to be fit or whose fit methods do nothing.

**Returns** True.

**property parameters**(*self*)

Returns the parameters which were used to initialize the component.

**save**(*self*, *file_path*, *pickle_protocol=cloudpickle.DEFAULT_PROTOCOL*)

Saves component at file path.

**Parameters**

- **file_path** (*str*) – Location to save file.
- **pickle_protocol** (*int*) – The pickle data stream format.

**transform**(*self*, *X*, *y=None*)

Transforms data X by applying the LSA pipeline.

**Parameters**

- **X** (*pd.DataFrame*) – The data to transform.
- **y** (*pd.Series, optional*) – Ignored.

**Returns**

**Transformed X. The original column is removed and replaced with two columns of the** format *LSA(original_column_name)[feature_number]*, where *feature_number* is 0 or 1.

**Return type** pd.DataFrame

**update_parameters**(*self*, *update_dict*, *reset_fit=True*)

Updates the parameter dictionary of the component.

**Parameters**

- **update_dict** (*dict*) – A dict of parameters to update.
- **reset_fit** (*bool, optional*) – If True, will set *_is_fitted* to False.

**class** evalml.pipelines.components.transformers.**NaturalLanguageFeaturizer**(*random_seed=0*, ***kwargs*)

Transformer that can automatically featurize text columns using featuretools' nlp_primitives.

Since models cannot handle non-numeric data, any text must be broken down into features that provide useful information about that text. This component splits each text column into several informative features: Diversity Score, Mean Characters per Word, Polarity Score, LSA (Latent Semantic Analysis), Number of Characters, and Number of Words. Calling transform on this component will replace any text columns in the given dataset with these numeric columns.

**Parameters** **random_seed** (*int*) – Seed for the random number generator. Defaults to 0.

**Attributes**

| | |
|---|---|
| **hyper-parameter_ranges** | {} |
| **modifies_features** | True |
| **modifies_target** | False |
| **name** | Natural Language Featurizer |
| **training_only** | False |

**Methods**

| | |
|---|---|
| *clone* | Constructs a new component with the same parameters and random state. |
| *default_parameters* | Returns the default parameters for this component. |
| *describe* | Describe a component and its parameters. |
| *fit* | Fits component to data. |
| *fit_transform* | Fits on X and transforms X. |
| *load* | Loads component at file path. |
| *needs_fitting* | Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances. |
| *parameters* | Returns the parameters which were used to initialize the component. |
| *save* | Saves component at file path. |
| *transform* | Transforms data X by creating new features using existing text columns. |
| *update_parameters* | Updates the parameter dictionary of the component. |

**clone**(*self*)

Constructs a new component with the same parameters and random state.

**Returns** A new instance of this component with identical parameters and random state.

**default_parameters**(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

**Returns** Default parameters for this component.

**Return type** dict

**describe**(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

**Parameters**

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description as dictionary in the format {"name": name, "parameters": parameters}

**Returns** Returns dictionary if return_dict is True, else None.

**Return type** None or dict

**fit**(*self*, *X*, *y=None*)

Fits component to data.

**Parameters**

- **X** (*pd.DataFrame or np.ndarray*) – The input training data of shape [n_samples, n_features]
- **y** (*pd.Series*) – The target training data of length [n_samples]

**Returns** self

**fit_transform**(*self*, *X*, *y=None*)

Fits on X and transforms X.

**Parameters**

- **X** (*pd.DataFrame*) – Data to fit and transform.
- **y** (*pd.Series*) – Target data.

**Returns** Transformed X.

**Return type** pd.DataFrame

**Raises** **MethodPropertyNotFoundError** – If transformer does not have a transform method or a component_obj that implements transform.

**static load**(*file_path*)

Loads component at file path.

**Parameters** **file_path** (*str*) – Location to load file.

**Returns** ComponentBase object

**needs_fitting**(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need to be fit or whose fit methods do nothing.

**Returns** True.

**property parameters**(*self*)

Returns the parameters which were used to initialize the component.

**save**(*self*, *file_path*, *pickle_protocol=cloudpickle.DEFAULT_PROTOCOL*)

Saves component at file path.

**Parameters**

- **file_path** (*str*) – Location to save file.

- **pickle_protocol** (*int*) – The pickle data stream format.

**transform**(*self*, *X*, *y=None*)

Transforms data X by creating new features using existing text columns.

**Parameters**

- **X** (*pd.DataFrame*) – The data to transform.
- **y** (*pd.Series, optional*) – Ignored.

**Returns** Transformed X

**Return type** pd.DataFrame

**update_parameters**(*self*, *update_dict*, *reset_fit=True*)

Updates the parameter dictionary of the component.

**Parameters**

- **update_dict** (*dict*) – A dict of parameters to update.
- **reset_fit** (*bool, optional*) – If True, will set *_is_fitted* to False.

**class** evalml.pipelines.components.transformers.**OneHotEncoder**(*top_n=10*, *features_to_encode=None*, *categories=None*, *drop='if_binary'*, *handle_unknown='ignore'*, *handle_missing='error'*, *random_seed=0*, ***kwargs*)

A transformer that encodes categorical features in a one-hot numeric array.

**Parameters**

- **top_n** (*int*) – Number of categories per column to encode. If None, all categories will be encoded. Otherwise, the *n* most frequent will be encoded and all others will be dropped. Defaults to 10.
- **features_to_encode** (*list[str]*) – List of columns to encode. All other columns will remain untouched. If None, all appropriate columns will be encoded. Defaults to None.
- **categories** (*list*) – A two dimensional list of categories, where *categories[i]* is a list of the categories for the column at index *i*. This can also be *None*, or *"auto"* if *top_n* is not None. Defaults to None.
- **drop** (*string, list*) – Method ("first" or "if_binary") to use to drop one category per feature. Can also be a list specifying which categories to drop for each feature. Defaults to 'if_binary'.
- **handle_unknown** (*string*) – Whether to ignore or error for unknown categories for a feature encountered during *fit* or *transform*. If either *top_n* or *categories* is used to limit the number of categories per column, this must be "ignore". Defaults to "ignore".
- **handle_missing** (*string*) – Options for how to handle missing (NaN) values encountered during *fit* or *transform*. If this is set to "as_category" and NaN values are within the *n* most frequent, "nan" values will be encoded as their own column. If this is set to "error", any missing values encountered will raise an error. Defaults to "error".
- **random_seed** (*int*) – Seed for the random number generator. Defaults to 0.

**Attributes**

| | |
|---|---|
| **hyper-parameter_ranges** | {} |
| **modifies_features** | True |
| **modifies_target** | False |
| **name** | One Hot Encoder |
| **training_only** | False |

**Methods**

| | |
|---|---|
| *categories* | Returns a list of the unique categories to be encoded for the particular feature, in order. |
| *clone* | Constructs a new component with the same parameters and random state. |
| *default_parameters* | Returns the default parameters for this component. |
| *describe* | Describe a component and its parameters. |
| *fit* | Fits the one-hot encoder component. |
| *fit_transform* | Fits on X and transforms X. |
| *get_feature_names* | Return feature names for the categorical features after fitting. |
| *load* | Loads component at file path. |
| *needs_fitting* | Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances. |
| *parameters* | Returns the parameters which were used to initialize the component. |
| *save* | Saves component at file path. |
| *transform* | One-hot encode the input data. |
| *update_parameters* | Updates the parameter dictionary of the component. |

**categories**(*self*, *feature_name*)

Returns a list of the unique categories to be encoded for the particular feature, in order.

**Parameters** **feature_name** (*str*) – The name of any feature provided to one-hot encoder during fit.

**Returns** The unique categories, in the same dtype as they were provided during fit.

**Return type** np.ndarray

**Raises** **ValueError** – If feature was not provided to one-hot encoder as a training feature.

**clone**(*self*)

Constructs a new component with the same parameters and random state.

**Returns** A new instance of this component with identical parameters and random state.

**default_parameters**(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

**Returns** Default parameters for this component.

**Return type** dict

**describe**(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

**Parameters**

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description as dictionary in the format {"name": name, "parameters": parameters}

**Returns** Returns dictionary if return_dict is True, else None.

**Return type** None or dict

**fit**(*self*, *X*, *y=None*)

Fits the one-hot encoder component.

**Parameters**

- **X** (*pd.DataFrame*) – The input training data of shape [n_samples, n_features].
- **y** (*pd.Series, optional*) – The target training data of length [n_samples].

**Returns** self

**Raises** **ValueError** – If encoding a column failed.

**fit_transform**(*self*, *X*, *y=None*)

Fits on X and transforms X.

**Parameters**

- **X** (*pd.DataFrame*) – Data to fit and transform.
- **y** (*pd.Series*) – Target data.

**Returns** Transformed X.

**Return type** pd.DataFrame

**Raises** **MethodPropertyNotFoundError** – If transformer does not have a transform method or a component_obj that implements transform.

**get_feature_names**(*self*)

Return feature names for the categorical features after fitting.

Feature names are formatted as {column name}_{category name}. In the event of a duplicate name, an integer will be added at the end of the feature name to distinguish it.

For example, consider a dataframe with a column called "A" and category "x_y" and another column called "A_x" with "y". In this example, the feature names would be "A_x_y" and "A_x_y_1".

**Returns** The feature names after encoding, provided in the same order as input_features.

**Return type** np.ndarray

**static load**(*file_path*)

Loads component at file path.

**Parameters** **file_path** (*str*) – Location to load file.

**Returns** ComponentBase object

**needs_fitting**(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need to be fit or whose fit methods do nothing.

**Returns** True.

**property parameters**(*self*)

Returns the parameters which were used to initialize the component.

**save**(*self*, *file_path*, *pickle_protocol=cloudpickle.DEFAULT_PROTOCOL*)

Saves component at file path.

**Parameters**

- **file_path** (*str*) – Location to save file.
- **pickle_protocol** (*int*) – The pickle data stream format.

**transform**(*self*, *X*, *y=None*)

One-hot encode the input data.

**Parameters**

- **X** (*pd.DataFrame*) – Features to one-hot encode.
- **y** (*pd.Series*) – Ignored.

**Returns** Transformed data, where each categorical feature has been encoded into numerical columns using one-hot encoding.

**Return type** pd.DataFrame

**update_parameters**(*self*, *update_dict*, *reset_fit=True*)

Updates the parameter dictionary of the component.

**Parameters**

- **update_dict** (*dict*) – A dict of parameters to update.
- **reset_fit** (*bool, optional*) – If True, will set *_is_fitted* to False.

**class** evalml.pipelines.components.transformers.**OrdinalEncoder**(*features_to_encode=None*, *categories=None*, *handle_unknown='error'*, *unknown_value=None*, *encoded_missing_value=None*, *random_seed=0*, ***kwargs*)

A transformer that encodes ordinal features as an array of ordinal integers representing the relative order of categories.

**Parameters**

- **features_to_encode** (*list[str]*) – List of columns to encode. All other columns will remain untouched. If None, all appropriate columns will be encoded. Defaults to None. The order of columns does not matter.
- **categories** (*dict[str, list[str]]*) – A dictionary mapping column names to their categories in the dataframes passed in at fit and transform. The order of categories specified for a column does not matter. Any category found in the data that is not present in categories will be handled as an unknown value. To not have unknown values raise an error, set handle_unknown to "use_encoded_value". Defaults to None.

- **handle_unknown** (*"error" or "use_encoded_value"*) – Whether to ignore or error for unknown categories for a feature encountered during *fit* or *transform*. When set to "error", an error will be raised when an unknown category is found. When set to "use_encoded_value", unknown categories will be encoded as the value given for the parameter unknown_value. Defaults to "error."
- **unknown_value** (*int or np.nan*) – The value to use for unknown categories seen during fit or transform. Required when the parameter handle_unknown is set to "use_encoded_value." The value has to be distinct from the values used to encode any of the categories in fit. Defaults to None.
- **encoded_missing_value** (*int or np.nan*) – The value to use for missing (null) values seen during fit or transform. Defaults to np.nan.
- **random_seed** (*int*) – Seed for the random number generator. Defaults to 0.

**Attributes**

| | |
|---|---|
| **hyper-parameter_ranges** | {} |
| **modifies_features** | True |
| **modifies_target** | False |
| **name** | Ordinal Encoder |
| **training_only** | False |

**Methods**

| | |
|---|---|
| *categories* | Returns a list of the unique categories to be encoded for the particular feature, in order. |
| *clone* | Constructs a new component with the same parameters and random state. |
| *default_parameters* | Returns the default parameters for this component. |
| *describe* | Describe a component and its parameters. |
| *fit* | Fits the ordinal encoder component. |
| *fit_transform* | Fits on X and transforms X. |
| *get_feature_names* | Return feature names for the ordinal features after fitting. |
| *load* | Loads component at file path. |
| *needs_fitting* | Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances. |
| *parameters* | Returns the parameters which were used to initialize the component. |
| *save* | Saves component at file path. |
| *transform* | Ordinally encode the input data. |
| *update_parameters* | Updates the parameter dictionary of the component. |

**categories**(*self*, *feature_name*)

Returns a list of the unique categories to be encoded for the particular feature, in order.

**Parameters** **feature_name** (*str*) – The name of any feature provided to ordinal encoder during fit.

**Returns** The unique categories, in the same dtype as they were provided during fit.

**Return type** np.ndarray

**Raises** **ValueError** – If feature was not provided to ordinal encoder as a training feature.

**clone**(*self*)

Constructs a new component with the same parameters and random state.

**Returns** A new instance of this component with identical parameters and random state.

**default_parameters**(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

**Returns** Default parameters for this component.

**Return type** dict

**describe**(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

**Parameters**

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description as dictionary in the format {"name": name, "parameters": parameters}

**Returns** Returns dictionary if return_dict is True, else None.

**Return type** None or dict

**fit**(*self*, *X*, *y=None*)

Fits the ordinal encoder component.

**Parameters**

- **X** (*pd.DataFrame*) – The input training data of shape [n_samples, n_features].
- **y** (*pd.Series, optional*) – The target training data of length [n_samples].

**Returns** self

**Raises**

- **ValueError** – If encoding a column failed.
- **TypeError** – If non-Ordinal columns are specified in features_to_encode.

**fit_transform**(*self*, *X*, *y=None*)

Fits on X and transforms X.

**Parameters**

- **X** (*pd.DataFrame*) – Data to fit and transform.
- **y** (*pd.Series*) – Target data.

**Returns** Transformed X.

**Return type** pd.DataFrame

**Raises** `MethodPropertyNotFoundError` – If transformer does not have a transform method or a component_obj that implements transform.

`get_feature_names`(*self*)

Return feature names for the ordinal features after fitting.

Feature names are formatted as {column name}_ordinal_encoding.

**Returns** The feature names after encoding, provided in the same order as input_features.

**Return type** np.ndarray

`static load`(*file_path*)

Loads component at file path.

**Parameters** `file_path` (*str*) – Location to load file.

**Returns** ComponentBase object

`needs_fitting`(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need to be fit or whose fit methods do nothing.

**Returns** True.

`property parameters`(*self*)

Returns the parameters which were used to initialize the component.

`save`(*self*, *file_path*, *pickle_protocol=cloudpickle.DEFAULT_PROTOCOL*)

Saves component at file path.

**Parameters**

- `file_path` (*str*) – Location to save file.
- `pickle_protocol` (*int*) – The pickle data stream format.

`transform`(*self*, *X*, *y=None*)

Ordinally encode the input data.

**Parameters**

- `X` (*pd.DataFrame*) – Features to encode.
- `y` (*pd.Series*) – Ignored.

**Returns** Transformed data, where each ordinal feature has been encoded into a numerical column where ordinal integers represent the relative order of categories.

**Return type** pd.DataFrame

`update_parameters`(*self*, *update_dict*, *reset_fit=True*)

Updates the parameter dictionary of the component.

**Parameters**

- `update_dict` (*dict*) – A dict of parameters to update.
- `reset_fit` (*bool, optional*) – If True, will set *_is_fitted* to False.

**class** evalml.pipelines.components.transformers.**Oversampler**(*sampling_ratio=0.25*, *sampling_ratio_dict=None*, *k_neighbors_default=5*, *n_jobs=- 1*, *random_seed=0*, ***kwargs*)

SMOTE Oversampler component. Will automatically select whether to use SMOTE, SMOTEN, or SMOTENC based on inputs to the component.

**Parameters**

- **sampling_ratio** (*float*) – This is the goal ratio of the minority to majority class, with range (0, 1]. A value of 0.25 means we want a 1:4 ratio of the minority to majority class after oversampling. We will create the a sampling dictionary using this ratio, with the keys corresponding to the class and the values responding to the number of samples. Defaults to 0.25.
- **sampling_ratio_dict** (*dict*) – A dictionary specifying the desired balanced ratio for each target value. For instance, in a binary case where class 1 is the minority, we could specify: *sampling_ratio_dict={0: 0.5, 1: 1}*, which means we would undersample class 0 to have twice the number of samples as class 1 (minority:majority ratio = 0.5), and don't sample class 1. Overrides sampling_ratio if provided. Defaults to None.
- **k_neighbors_default** (*int*) – The number of nearest neighbors used to construct synthetic samples. This is the default value used, but the actual k_neighbors value might be smaller if there are less samples. Defaults to 5.
- **n_jobs** (*int*) – The number of CPU cores to use. Defaults to -1.
- **random_seed** (*int*) – The seed to use for random sampling. Defaults to 0.

**Attributes**

| | |
|---|---|
| **hyper-parameter_ranges** | None |
| **modifies_features** | True |
| **modifies_target** | True |
| **name** | Oversampler |
| **training_only** | True |

**Methods**

| | |
|---|---|
| *clone* | Constructs a new component with the same parameters and random state. |
| *default_parameters* | Returns the default parameters for this component. |
| *describe* | Describe a component and its parameters. |
| *fit* | Fits oversampler to data. |
| *fit_transform* | Fit and transform data using the sampler component. |
| *load* | Loads component at file path. |
| *needs_fitting* | Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances. |
| *parameters* | Returns the parameters which were used to initialize the component. |
| *save* | Saves component at file path. |
| *transform* | Transforms the input data by Oversampling the data. |
| *update_parameters* | Updates the parameter dictionary of the component. |

**clone**(*self*)

Constructs a new component with the same parameters and random state.

**Returns** A new instance of this component with identical parameters and random state.

**default_parameters**(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

**Returns** Default parameters for this component.

**Return type** dict

**describe**(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

**Parameters**

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description as dictionary in the format {"name": name, "parameters": parameters}

**Returns** Returns dictionary if return_dict is True, else None.

**Return type** None or dict

**fit**(*self*, *X*, *y*)

Fits oversampler to data.

**Parameters**

- **X** (*pd.DataFrame*) – The input training data of shape [n_samples, n_features].
- **y** (*pd.Series, optional*) – The target training data of length [n_samples].

**Returns** self

**fit_transform**(*self*, *X*, *y*)

Fit and transform data using the sampler component.

**Parameters**

- **X** (*pd.DataFrame*) – The input training data of shape [n_samples, n_features].

- **y** (*pd.Series, optional*) – The target training data of length [n_samples].

**Returns** Transformed data.

**Return type** (pd.DataFrame, pd.Series)

**static load**(*file_path*)

Loads component at file path.

**Parameters** **file_path** (*str*) – Location to load file.

**Returns** ComponentBase object

**needs_fitting**(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need to be fit or whose fit methods do nothing.

**Returns** True.

**property parameters**(*self*)

Returns the parameters which were used to initialize the component.

**save**(*self*, *file_path*, *pickle_protocol=cloudpickle.DEFAULT_PROTOCOL*)

Saves component at file path.

**Parameters**

- **file_path** (*str*) – Location to save file.
- **pickle_protocol** (*int*) – The pickle data stream format.

**transform**(*self*, *X*, *y=None*)

Transforms the input data by Oversampling the data.

**Parameters**

- **X** (*pd.DataFrame*) – Training features.
- **y** (*pd.Series*) – Target.

**Returns** Transformed features and target.

**Return type** pd.DataFrame, pd.Series

**update_parameters**(*self*, *update_dict*, *reset_fit=True*)

Updates the parameter dictionary of the component.

**Parameters**

- **update_dict** (*dict*) – A dict of parameters to update.
- **reset_fit** (*bool, optional*) – If True, will set *_is_fitted* to False.

**class** evalml.pipelines.components.transformers.**PCA**(*variance=0.95*, *n_components=None*, *random_seed=0*, ***kwargs*)

Reduces the number of features by using Principal Component Analysis (PCA).

**Parameters**

- **variance** (*float*) – The percentage of the original data variance that should be preserved when reducing the number of features. Defaults to 0.95.
- **n_components** (*int*) – The number of features to maintain after computing SVD. Defaults to None, but will override variance variable if set.

- **random_seed** (*int*) – Seed for the random number generator. Defaults to 0.

**Attributes**

| | |
|---|---|
| **hyper-parameter_ranges** | Real(0.25, 1)}:type: {"variance" |
| **modifies_features** | True |
| **modifies_target** | False |
| **name** | PCA Transformer |
| **training_only** | False |

**Methods**

| | |
|---|---|
| *clone* | Constructs a new component with the same parameters and random state. |
| *default_parameters* | Returns the default parameters for this component. |
| *describe* | Describe a component and its parameters. |
| *fit* | Fits the PCA component. |
| *fit_transform* | Fit and transform data using the PCA component. |
| *load* | Loads component at file path. |
| *needs_fitting* | Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances. |
| *parameters* | Returns the parameters which were used to initialize the component. |
| *save* | Saves component at file path. |
| *transform* | Transform data using fitted PCA component. |
| *update_parameters* | Updates the parameter dictionary of the component. |

**clone**(*self*)

Constructs a new component with the same parameters and random state.

**Returns** A new instance of this component with identical parameters and random state.

**default_parameters**(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

**Returns** Default parameters for this component.

**Return type** dict

**describe**(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

**Parameters**

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description as dictionary in the format {"name": name, "parameters": parameters}

**Returns** Returns dictionary if return_dict is True, else None.

**Return type** None or dict

`fit`(*self*, *X*, *y=None*)

Fits the PCA component.

**Parameters**

- **X** (*pd.DataFrame*) – The input training data of shape [n_samples, n_features].
- **y** (*pd.Series, optional*) – The target training data of length [n_samples].

**Returns** self

**Raises** **ValueError** – If input data is not all numeric.

`fit_transform`(*self*, *X*, *y=None*)

Fit and transform data using the PCA component.

**Parameters**

- **X** (*pd.DataFrame*) – The input training data of shape [n_samples, n_features].
- **y** (*pd.Series, optional*) – The target training data of length [n_samples].

**Returns** Transformed data.

**Return type** pd.DataFrame

**Raises** **ValueError** – If input data is not all numeric.

`static load`(*file_path*)

Loads component at file path.

**Parameters** **file_path** (*str*) – Location to load file.

**Returns** ComponentBase object

`needs_fitting`(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need to be fit or whose fit methods do nothing.

**Returns** True.

`property parameters`(*self*)

Returns the parameters which were used to initialize the component.

`save`(*self*, *file_path*, *pickle_protocol=cloudpickle.DEFAULT_PROTOCOL*)

Saves component at file path.

**Parameters**

- **file_path** (*str*) – Location to save file.
- **pickle_protocol** (*int*) – The pickle data stream format.

`transform`(*self*, *X*, *y=None*)

Transform data using fitted PCA component.

**Parameters**

- **X** (*pd.DataFrame*) – The input training data of shape [n_samples, n_features].
- **y** (*pd.Series, optional*) – The target training data of length [n_samples].

**Returns** Transformed data.

**Return type** pd.DataFrame

**Raises** `ValueError` – If input data is not all numeric.

`update_parameters`(*self*, *update_dict*, *reset_fit=True*)

Updates the parameter dictionary of the component.

**Parameters**

- `update_dict` (*dict*) – A dict of parameters to update.
- `reset_fit` (*bool, optional*) – If True, will set *_is_fitted* to False.

`class evalml.pipelines.components.transformers.PerColumnImputer`(*impute_strategies=None*, *random_seed=0*, ***kwargs*)

Imputes missing data according to a specified imputation strategy per column.

**Parameters**

- `impute_strategies` (*dict*) – Column and {"impute_strategy": strategy, "fill_value":value} pairings. Valid values for impute strategy include "mean", "median", "most_frequent", "constant" for numerical data, and "most_frequent", "constant" for object data types. Defaults to None, which uses "most_frequent" for all columns. When impute_strategy == "constant", fill_value is used to replace missing data. When None, uses 0 when imputing numerical data and "missing_value" for strings or object data types.
- `random_seed` (*int*) – Seed for the random number generator. Defaults to 0.

**Attributes**

| | |
|---|---|
| **hyper-parameter_ranges** | {} |
| **modifies_features** | True |
| **modifies_target** | False |
| **name** | Per Column Imputer |
| **training_only** | False |

**Methods**

| | |
|---|---|
| *clone* | Constructs a new component with the same parameters and random state. |
| *default_parameters* | Returns the default parameters for this component. |
| *describe* | Describe a component and its parameters. |
| *fit* | Fits imputers on input data. |
| *fit_transform* | Fits on X and transforms X. |
| *load* | Loads component at file path. |
| *needs_fitting* | Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances. |
| *parameters* | Returns the parameters which were used to initialize the component. |
| *save* | Saves component at file path. |
| *transform* | Transforms input data by imputing missing values. |
| *update_parameters* | Updates the parameter dictionary of the component. |

**clone**(*self*)

Constructs a new component with the same parameters and random state.

**Returns** A new instance of this component with identical parameters and random state.

**default_parameters**(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

**Returns** Default parameters for this component.

**Return type** dict

**describe**(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

**Parameters**

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description as dictionary in the format {"name": name, "parameters": parameters}

**Returns** Returns dictionary if return_dict is True, else None.

**Return type** None or dict

**fit**(*self*, *X*, *y=None*)

Fits imputers on input data.

**Parameters**

- **X** (*pd.DataFrame or np.ndarray*) – The input training data of shape [n_samples, n_features] to fit.
- **y** (*pd.Series, optional*) – The target training data of length [n_samples]. Ignored.

**Returns** self

**fit_transform**(*self*, *X*, *y=None*)

Fits on X and transforms X.

**Parameters**

- **X** (*pd.DataFrame*) – Data to fit and transform.
- **y** (*pd.Series*) – Target data.

**Returns** Transformed X.

**Return type** pd.DataFrame

**Raises** **MethodPropertyNotFoundError** – If transformer does not have a transform method or a component_obj that implements transform.

**static load**(*file_path*)

Loads component at file path.

**Parameters** **file_path** (*str*) – Location to load file.

**Returns** ComponentBase object

**needs_fitting**(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need to be fit or whose fit methods do nothing.

**Returns** True.

**property parameters**(*self*)

Returns the parameters which were used to initialize the component.

**save**(*self*, *file_path*, *pickle_protocol=cloudpickle.DEFAULT_PROTOCOL*)

Saves component at file path.

**Parameters**

- **file_path** (*str*) – Location to save file.
- **pickle_protocol** (*int*) – The pickle data stream format.

**transform**(*self*, *X*, *y=None*)

Transforms input data by imputing missing values.

**Parameters**

- **X** (*pd.DataFrame or np.ndarray*) – The input training data of shape [n_samples, n_features] to transform.
- **y** (*pd.Series, optional*) – The target training data of length [n_samples]. Ignored.

**Returns** Transformed X

**Return type** pd.DataFrame

**update_parameters**(*self*, *update_dict*, *reset_fit=True*)

Updates the parameter dictionary of the component.

**Parameters**

- **update_dict** (*dict*) – A dict of parameters to update.
- **reset_fit** (*bool, optional*) – If True, will set *_is_fitted* to False.

**class** evalml.pipelines.components.transformers.**PolynomialDecomposer**(*time_index: str = None*, *degree: int = 1*, *period: int = - 1*, *random_seed: int = 0*, ***kwargs*)

Removes trends and seasonality from time series by fitting a polynomial and moving average to the data.

**Scikit-learn's PolynomialForecaster is used to generate the additive trend portion of the target data. A polynomial** will be fit to the data during fit. That additive polynomial trend will be removed during fit so that statsmodel's seasonal_decompose can determine the addititve seasonality of the data by using rolling averages over the series' inferred periodicity.

For example, daily time series data will generate rolling averages over the first week of data, normalize out the mean and return those 7 averages repeated over the entire length of the given series. Those seven averages, repeated as many times as necessary to match the length of the given target data, will be used as the seasonal signal of the data.

**Parameters**

- **time_index** (*str*) – Specifies the name of the column in X that provides the datetime objects. Defaults to None.
- **degree** (*int*) – Degree for the polynomial. If 1, linear model is fit to the data. If 2, quadratic model is fit, etc. Defaults to 1.
- **period** (*int*) – The number of entries in the time series data that corresponds to one period of a cyclic signal. For instance, if data is known to possess a weekly seasonal signal, and if the data is daily data, period should be 7. For daily data with a yearly seasonal signal, period should be 365. Defaults to -1, which uses the statsmodels libarary's freq_to_period function. https://github.com/statsmodels/statsmodels/blob/main/statsmodels/tsa/tsatools.py
- **random_seed** (*int*) – Seed for the random number generator. Defaults to 0.

**Attributes**

| | |
|---|---|
| **hyper-parameter_ranges** | { "degree": Integer(1, 3)} |
| **invalid_frequencies** | [] |
| **modifies_features** | False |
| **modifies_target** | True |
| **name** | Polynomial Decomposer |
| **needs_fitting** | True |
| **training_only** | False |

**Methods**

| | |
|---|---|
| *clone* | Constructs a new component with the same parameters and random state. |
| *default_parameters* | Returns the default parameters for this component. |
| *describe* | Describe a component and its parameters. |
| *determine_periodicity* | Function that uses autocorrelative methods to determine the likely most signficant period of the seasonal signal. |
| *fit* | Fits the PolynomialDecomposer and determine the seasonal signal. |
| *fit_transform* | Removes fitted trend and seasonality from target variable. |
| *get_trend_dataframe* | Return a list of dataframes with 4 columns: signal, trend, seasonality, residual. |
| *inverse_transform* | Adds back fitted trend and seasonality to target variable. |
| *is_freq_valid* | Determines if the given string represents a valid frequency for this decomposer. |
| *load* | Loads component at file path. |
| *parameters* | Returns the parameters which were used to initialize the component. |
| *plot_decomposition* | Plots the decomposition of the target signal. |
| *save* | Saves component at file path. |
| *set_period* | Function to set the component's seasonal period based on the target's seasonality. |
| *transform* | Transforms the target data by removing the polynomial trend and rolling average seasonality. |
| *update_parameters* | Updates the parameter dictionary of the component. |

**clone**(*self*)

Constructs a new component with the same parameters and random state.

**Returns** A new instance of this component with identical parameters and random state.

**default_parameters**(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

**Returns** Default parameters for this component.

**Return type** dict

**describe**(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

**Parameters**

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description as dictionary in the format {"name": name, "parameters": parameters}

**Returns** Returns dictionary if return_dict is True, else None.

**Return type** None or dict

**classmethod determine_periodicity**(*cls*, *X: pandas.DataFrame*, *y: pandas.Series*, *acf_threshold: float = 0.01*, *rel_max_order: int = 5*)

Function that uses autocorrelative methods to determine the likely most signficant period of the seasonal signal.

**Parameters**

- **X** (*pandas.DataFrame*) – The feature data of the time series problem.
- **y** (*pandas.Series*) – The target data of a time series problem.
- **acf_threshold** (*float*) – The threshold for the autocorrelation function to determine the period. Any values below the threshold are considered to be 0 and will not be considered for the period. Defaults to 0.01.
- **rel_max_order** (*int*) – The order of the relative maximum to determine the period. Defaults to 5.

**Returns**

**The integer number of entries in time series data over which the seasonal part of the target data** repeats. If the time series data is in days, then this is the number of days that it takes the target's seasonal signal to repeat. Note: the target data can contain multiple seasonal signals. This function will only return the stronger. E.g. if the target has both weekly and yearly seasonality, the function may return either "7" or "365", depending on which seasonality is more strongly autocorrelated. If no period is detected, returns None.

**Return type** int

**fit**(*self*, *X: pandas.DataFrame*, *y: pandas.Series = None*) → *PolynomialDecomposer*

Fits the PolynomialDecomposer and determine the seasonal signal.

Currently only fits the polynomial detrender. The seasonality is determined by removing the trend from the signal and using statsmodels' seasonal_decompose(). Both the trend and seasonality are currently assumed to be additive.

**Parameters**

- **X** (*pd.DataFrame, optional*) – Conditionally used to build datetime index.
- **y** (*pd.Series*) – Target variable to detrend and deseasonalize.

**Returns** self

**Raises**

- **NotImplementedError** – If the input data has a frequency of "month-begin". This isn't supported by statsmodels decompose as the freqstr "MS" is misinterpreted as milliseconds.
- **ValueError** – If y is None.
- **ValueError** – If target data doesn't have DatetimeIndex AND no Datetime features in features data

**fit_transform**(*self*, *X: pandas.DataFrame*, *y: pandas.Series = None*) → tuple[pandas.DataFrame, pandas.Series]

Removes fitted trend and seasonality from target variable.

**Parameters**

- **X** (*pd.DataFrame, optional*) – Ignored.
- **y** (*pd.Series*) – Target variable to detrend and deseasonalize.

**Returns**

**The first element are the input features returned without modification.** The second element is the target variable y with the fitted trend removed.

**Return type** tuple of pd.DataFrame, pd.Series

`get_trend_dataframe`(*self*, *X: pandas.DataFrame*, *y: pandas.Series*) → list[pandas.DataFrame]

Return a list of dataframes with 4 columns: signal, trend, seasonality, residual.

Scikit-learn's PolynomialForecaster is used to generate the trend portion of the target data. statsmodel's seasonal_decompose is used to generate the seasonality of the data.

**Parameters**

- **X** (*pd.DataFrame*) – Input data with time series data in index.
- **y** (*pd.Series or pd.DataFrame*) – Target variable data provided as a Series for univariate problems or a DataFrame for multivariate problems.

**Returns**

**Each DataFrame contains the columns "signal", "trend", "seasonality" and "residual,"** with the latter 3 column values being the decomposed elements of the target data. The "signal" column is simply the input target signal but reindexed with a datetime index to match the input features.

**Return type** list of pd.DataFrame

**Raises**

- **TypeError** – If X does not have time-series data in the index.
- **ValueError** – If time series index of X does not have an inferred frequency.
- **ValueError** – If the forecaster associated with the detrender has not been fit yet.
- **TypeError** – If y is not provided as a pandas Series or DataFrame.

`inverse_transform`(*self*, *y_t: pandas.Series*) → tuple[pandas.DataFrame, pandas.Series]

Adds back fitted trend and seasonality to target variable.

The polynomial trend is added back into the signal, calling the detrender's inverse_transform(). Then, the seasonality is projected forward to and added back into the signal.

**Parameters** **y_t** (*pd.Series*) – Target variable.

**Returns**

**The first element are the input features returned without modification.** The second element is the target variable y with the trend and seasonality added back in.

**Return type** tuple of pd.DataFrame, pd.Series

**Raises** **ValueError** – If y is None.

`classmethod is_freq_valid`(*cls*, *freq: str*)

Determines if the given string represents a valid frequency for this decomposer.

**Parameters** **freq** (*str*) – A frequency to validate. See the pandas docs at https://pandas.pydata.org/pandas-docs/stable/user_guide/timeseries.html#offset-aliases for options.

**Returns** boolean representing whether the frequency is valid or not.

`static load`(*file_path*)

Loads component at file path.

**Parameters** **file_path** (*str*) – Location to load file.

**Returns** ComponentBase object

**property parameters**(*self*)

Returns the parameters which were used to initialize the component.

**plot_decomposition**(*self*, *X: pandas.DataFrame*, *y: pandas.Series*, *show: bool = False*) → tuple[matplotlib.pyplot.Figure, list]

Plots the decomposition of the target signal.

**Parameters**

- **X** (*pd.DataFrame*) – Input data with time series data in index.
- **y** (*pd.Series or pd.DataFrame*) – Target variable data provided as a Series for univariate problems or a DataFrame for multivariate problems.
- **show** (*bool*) – Whether to display the plot or not. Defaults to False.

**Returns**

**The figure and axes that have the decompositions** plotted on them

**Return type** matplotlib.pyplot.Figure, list[matplotlib.pyplot.Axes]

**save**(*self*, *file_path*, *pickle_protocol=cloudpickle.DEFAULT_PROTOCOL*)

Saves component at file path.

**Parameters**

- **file_path** (*str*) – Location to save file.
- **pickle_protocol** (*int*) – The pickle data stream format.

**set_period**(*self*, *X: pandas.DataFrame*, *y: pandas.Series*, *acf_threshold: float = 0.01*, *rel_max_order: int = 5*)

Function to set the component's seasonal period based on the target's seasonality.

**Parameters**

- **X** (*pandas.DataFrame*) – The feature data of the time series problem.
- **y** (*pandas.Series*) – The target data of a time series problem.
- **acf_threshold** (*float*) – The threshold for the autocorrelation function to determine the period. Any values below the threshold are considered to be 0 and will not be considered for the period. Defaults to 0.01.
- **rel_max_order** (*int*) – The order of the relative maximum to determine the period. Defaults to 5.

**transform**(*self*, *X: pandas.DataFrame*, *y: pandas.Series = None*) → tuple[pandas.DataFrame, pandas.Series]

Transforms the target data by removing the polynomial trend and rolling average seasonality.

Applies the fit polynomial detrender to the target data, removing the additive polynomial trend. Then, utilizes the first period's worth of seasonal data determined in the .fit() function to extrapolate the seasonal signal of the data to be transformed. This seasonal signal is also assumed to be additive and is removed.

**Parameters**

- **X** (*pd.DataFrame, optional*) – Conditionally used to build datetime index.
- **y** (*pd.Series*) – Target variable to detrend and deseasonalize.

**Returns**

**The input features are returned without modification. The target** variable y is detrended and deseasonalized.

**Return type** tuple of pd.DataFrame, pd.Series

**Raises** `ValueError` – If target data doesn't have DatetimeIndex AND no Datetime features in features data

`update_parameters`(*self*, *update_dict*, *reset_fit=True*)

Updates the parameter dictionary of the component.

**Parameters**

- `update_dict` (*dict*) – A dict of parameters to update.
- `reset_fit` (*bool, optional*) – If True, will set *_is_fitted* to False.

**class** `evalml.pipelines.components.transformers.`**`ReplaceNullableTypes`**(*random_seed=0*, ***kwargs*)

Transformer to replace features with the new nullable dtypes with a dtype that is compatible in EvalML.

**Attributes**

| | |
|---|---|
| **hyper-parameter_ranges** | None |
| **modifies_features** | True |
| **modifies_target** | {} |
| **name** | Replace Nullable Types Transformer |
| **training_only** | False |

**Methods**

| | |
|---|---|
| *clone* | Constructs a new component with the same parameters and random state. |
| *default_parameters* | Returns the default parameters for this component. |
| *describe* | Describe a component and its parameters. |
| *fit* | Fits component to data. |
| *fit_transform* | Substitutes non-nullable types for the new pandas nullable types in the data and target data. |
| *load* | Loads component at file path. |
| *needs_fitting* | Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances. |
| *parameters* | Returns the parameters which were used to initialize the component. |
| *save* | Saves component at file path. |
| *transform* | Transforms data by replacing columns that contain nullable types with the appropriate replacement type. |
| *update_parameters* | Updates the parameter dictionary of the component. |

`clone`(*self*)

Constructs a new component with the same parameters and random state.

**Returns** A new instance of this component with identical parameters and random state.

`default_parameters`(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

**Returns** Default parameters for this component.

**Return type** dict

`describe`(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

**Parameters**

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description as dictionary in the format {"name": name, "parameters": parameters}

**Returns** Returns dictionary if return_dict is True, else None.

**Return type** None or dict

`fit`(*self*, *X*, *y=None*)

Fits component to data.

**Parameters**

- **X** (*pd.DataFrame*) – The input training data of shape [n_samples, n_features].
- **y** (*pd.Series, optional*) – The target training data of length [n_samples].

**Returns** self

`fit_transform`(*self*, *X*, *y=None*)

Substitutes non-nullable types for the new pandas nullable types in the data and target data.

**Parameters**

- **X** (*pd.DataFrame, optional*) – Input features.
- **y** (*pd.Series*) – Target data.

**Returns** The input features and target data with the non-nullable types set.

**Return type** tuple of pd.DataFrame, pd.Series

`static load`(*file_path*)

Loads component at file path.

**Parameters** **file_path** (*str*) – Location to load file.

**Returns** ComponentBase object

`needs_fitting`(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need to be fit or whose fit methods do nothing.

**Returns** True.

`property parameters`(*self*)

Returns the parameters which were used to initialize the component.

**save**(*self*, *file_path*, *pickle_protocol=cloudpickle.DEFAULT_PROTOCOL*)

Saves component at file path.

**Parameters**

- **file_path** (*str*) – Location to save file.
- **pickle_protocol** (*int*) – The pickle data stream format.

**transform**(*self*, *X*, *y=None*)

Transforms data by replacing columns that contain nullable types with the appropriate replacement type.

"float64" for nullable integers and "category" for nullable booleans.

**Parameters**

- **X** (*pd.DataFrame*) – Data to transform
- **y** (*pd.Series, optional*) – Target data to transform

**Returns** Transformed X pd.Series: Transformed y

**Return type** pd.DataFrame

**update_parameters**(*self*, *update_dict*, *reset_fit=True*)

Updates the parameter dictionary of the component.

**Parameters**

- **update_dict** (*dict*) – A dict of parameters to update.
- **reset_fit** (*bool, optional*) – If True, will set *_is_fitted* to False.

**class** evalml.pipelines.components.transformers.**RFClassifierRFESelector**(*step=0.2*, *min_features_to_select=1*, *cv=None*, *scoring=None*, *n_jobs=- 1*, *n_estimators=10*, *max_depth=None*, *random_seed=0*, ***kwargs*)

Selects relevant features using recursive feature elimination with a Random Forest Classifier.

**Parameters**

- **step** (*int, float*) – The number of features to eliminate in each iteration. If an integer is specified this will represent the number of features to eliminate. If a float is specified this represents the percentage of features to eliminate each iteration. The last iteration may drop fewer than this number of features in order to satisfy the min_features_to_select constraint. Defaults to 0.2.
- **min_features_to_select** (*int*) – The minimum number of features to return. Defaults to 1.
- **cv** (*int or None*) – Number of folds to use for the cross-validation splitting strategy. Defaults to None which will use 5 folds.
- **scoring** (*str, callable or None*) – A string or scorer callable object to specify the scoring method.
- **n_jobs** (*int or None*) – Number of jobs to run in parallel. -1 uses all processes. Defaults to -1.

- **n_estimators** (*int*) – The number of trees in the forest. Defaults to 10.
- **max_depth** (*int*) – Maximum tree depth for base learners. Defaults to None.
- **random_seed** (*int*) – Seed for the random number generator. Defaults to 0.

**Attributes**

| | |
|---|---|
| **hyper-parameter_ranges** | { "step": Real(0.05, 0.25)} |
| **modifies_features** | True |
| **modifies_target** | False |
| **name** | RFE Selector with RF Classifier |
| **training_only** | False |

**Methods**

| | |
|---|---|
| *clone* | Constructs a new component with the same parameters and random state. |
| *default_parameters* | Returns the default parameters for this component. |
| *describe* | Describe a component and its parameters. |
| *fit* | Fits component to data. |
| *fit_transform* | Fit and transform data using the feature selector. |
| *get_names* | Get names of selected features. |
| *load* | Loads component at file path. |
| *needs_fitting* | Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances. |
| *parameters* | Returns the parameters which were used to initialize the component. |
| *save* | Saves component at file path. |
| *transform* | Transforms input data by selecting features. If the component_obj does not have a transform method, will raise an MethodPropertyNotFoundError exception. |
| *update_parameters* | Updates the parameter dictionary of the component. |

**clone**(*self*)

Constructs a new component with the same parameters and random state.

**Returns** A new instance of this component with identical parameters and random state.

**default_parameters**(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

**Returns** Default parameters for this component.

**Return type** dict

**describe**(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

**Parameters**

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description as dictionary in the format {"name": name, "parameters": parameters}

**Returns** Returns dictionary if return_dict is True, else None.

**Return type** None or dict

**fit**(*self*, *X*, *y=None*)

Fits component to data.

**Parameters**

- **X** (*pd.DataFrame*) – The input training data of shape [n_samples, n_features]
- **y** (*pd.Series, optional*) – The target training data of length [n_samples]

**Returns** self

**Raises** **MethodPropertyNotFoundError** – If component does not have a fit method or a component_obj that implements fit.

**fit_transform**(*self*, *X*, *y=None*)

Fit and transform data using the feature selector.

**Parameters**

- **X** (*pd.DataFrame*) – The input training data of shape [n_samples, n_features].
- **y** (*pd.Series, optional*) – The target training data of length [n_samples].

**Returns** Transformed data.

**Return type** pd.DataFrame

**get_names**(*self*)

Get names of selected features.

**Returns** List of the names of features selected.

**Return type** list[str]

**static load**(*file_path*)

Loads component at file path.

**Parameters** **file_path** (*str*) – Location to load file.

**Returns** ComponentBase object

**needs_fitting**(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need to be fit or whose fit methods do nothing.

**Returns** True.

**property parameters**(*self*)

Returns the parameters which were used to initialize the component.

**save**(*self*, *file_path*, *pickle_protocol=cloudpickle.DEFAULT_PROTOCOL*)

Saves component at file path.

**Parameters**

- **file_path** (*str*) – Location to save file.
- **pickle_protocol** (*int*) – The pickle data stream format.

**transform**(*self*, *X*, *y=None*)

Transforms input data by selecting features. If the component_obj does not have a transform method, will raise an MethodPropertyNotFoundError exception.

**Parameters**

- **X** (*pd.DataFrame*) – Data to transform.
- **y** (*pd.Series, optional*) – Target data. Ignored.

**Returns** Transformed X

**Return type** pd.DataFrame

**Raises** **MethodPropertyNotFoundError** – If feature selector does not have a transform method or a component_obj that implements transform

**update_parameters**(*self*, *update_dict*, *reset_fit=True*)

Updates the parameter dictionary of the component.

**Parameters**

- **update_dict** (*dict*) – A dict of parameters to update.
- **reset_fit** (*bool, optional*) – If True, will set *_is_fitted* to False.

class evalml.pipelines.components.transformers.**RFClassifierSelectFromModel**(*number_features=None*, *n_estimators=10*, *max_depth=None*, *percent_features=0.5*, *threshold='median'*, *n_jobs=-1*, *random_seed=0*, ***kwargs*)

Selects top features based on importance weights using a Random Forest classifier.

**Parameters**

- **number_features** (*int*) – The maximum number of features to select. If both percent_features and number_features are specified, take the greater number of features. Defaults to None.
- **n_estimators** (*int*) – The number of trees in the forest. Defaults to 10.
- **max_depth** (*int*) – Maximum tree depth for base learners. Defaults to None.
- **percent_features** (*float*) – Percentage of features to use. If both percent_features and number_features are specified, take the greater number of features. Defaults to 0.5.
- **threshold** (*string or float*) – The threshold value to use for feature selection. Features whose importance is greater or equal are kept while the others are discarded. If "median", then the threshold value is the median of the feature importances. A scaling factor (e.g., "1.25*mean") may also be used. Defaults to median.

- **n_jobs** (*int or None*) – Number of jobs to run in parallel. -1 uses all processes. Defaults to -1.
- **random_seed** (*int*) – Seed for the random number generator. Defaults to 0.

**Attributes**

| | |
|---|---|
| **hyper-parameter_ranges** | { "percent_features": Real(0.01, 1), "threshold": ["mean", "median"],} |
| **modifies_features** | True |
| **modifies_target** | False |
| **name** | RF Classifier Select From Model |
| **training_only** | False |

**Methods**

| | |
|---|---|
| *clone* | Constructs a new component with the same parameters and random state. |
| *default_parameters* | Returns the default parameters for this component. |
| *describe* | Describe a component and its parameters. |
| *fit* | Fits component to data. |
| *fit_transform* | Fit and transform data using the feature selector. |
| *get_names* | Get names of selected features. |
| *load* | Loads component at file path. |
| *needs_fitting* | Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances. |
| *parameters* | Returns the parameters which were used to initialize the component. |
| *save* | Saves component at file path. |
| *transform* | Transforms input data by selecting features. If the component_obj does not have a transform method, will raise an MethodPropertyNotFoundError exception. |
| *update_parameters* | Updates the parameter dictionary of the component. |

**clone**(*self*)

Constructs a new component with the same parameters and random state.

**Returns** A new instance of this component with identical parameters and random state.

**default_parameters**(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

**Returns** Default parameters for this component.

**Return type** dict

**describe**(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

**Parameters**

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description as dictionary in the format {"name": name, "parameters": parameters}

**Returns** Returns dictionary if return_dict is True, else None.

**Return type** None or dict

**fit**(*self*, *X*, *y=None*)

Fits component to data.

**Parameters**

- **X** (*pd.DataFrame*) – The input training data of shape [n_samples, n_features]
- **y** (*pd.Series, optional*) – The target training data of length [n_samples]

**Returns** self

**Raises** **MethodPropertyNotFoundError** – If component does not have a fit method or a component_obj that implements fit.

**fit_transform**(*self*, *X*, *y=None*)

Fit and transform data using the feature selector.

**Parameters**

- **X** (*pd.DataFrame*) – The input training data of shape [n_samples, n_features].
- **y** (*pd.Series, optional*) – The target training data of length [n_samples].

**Returns** Transformed data.

**Return type** pd.DataFrame

**get_names**(*self*)

Get names of selected features.

**Returns** List of the names of features selected.

**Return type** list[str]

**static load**(*file_path*)

Loads component at file path.

**Parameters** **file_path** (*str*) – Location to load file.

**Returns** ComponentBase object

**needs_fitting**(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need to be fit or whose fit methods do nothing.

**Returns** True.

**property parameters**(*self*)

Returns the parameters which were used to initialize the component.

**save**(*self*, *file_path*, *pickle_protocol=cloudpickle.DEFAULT_PROTOCOL*)

Saves component at file path.

**Parameters**

- **file_path** (*str*) – Location to save file.
- **pickle_protocol** (*int*) – The pickle data stream format.

**transform**(*self*, *X*, *y=None*)

Transforms input data by selecting features. If the component_obj does not have a transform method, will raise an MethodPropertyNotFoundError exception.

**Parameters**

- **X** (*pd.DataFrame*) – Data to transform.
- **y** (*pd.Series, optional*) – Target data. Ignored.

**Returns** Transformed X

**Return type** pd.DataFrame

**Raises** **MethodPropertyNotFoundError** – If feature selector does not have a transform method or a component_obj that implements transform

**update_parameters**(*self*, *update_dict*, *reset_fit=True*)

Updates the parameter dictionary of the component.

**Parameters**

- **update_dict** (*dict*) – A dict of parameters to update.
- **reset_fit** (*bool, optional*) – If True, will set *_is_fitted* to False.

**class** evalml.pipelines.components.transformers.**RFRegressorRFESelector**(*step=0.2*, *min_features_to_select=1*, *cv=None*, *scoring=None*, *n_jobs=- 1*, *n_estimators=10*, *max_depth=None*, *random_seed=0*, ***kwargs*)

Selects relevant features using recursive feature elimination with a Random Forest Regressor.

**Parameters**

- **step** (*int, float*) – The number of features to eliminate in each iteration. If an integer is specified this will represent the number of features to eliminate. If a float is specified this represents the percentage of features to eliminate each iteration. The last iteration may drop fewer than this number of features in order to satisfy the min_features_to_select constraint. Defaults to 0.2.
- **min_features_to_select** (*int*) – The minimum number of features to return. Defaults to 1.
- **cv** (*int or None*) – Number of folds to use for the cross-validation splitting strategy. Defaults to None which will use 5 folds.
- **scoring** (*str, callable or None*) – A string or scorer callable object to specify the scoring method.

- **n_jobs** (*int or None*) – Number of jobs to run in parallel. -1 uses all processes. Defaults to -1.
- **n_estimators** (*int*) – The number of trees in the forest. Defaults to 10.
- **max_depth** (*int*) – Maximum tree depth for base learners. Defaults to None.
- **random_seed** (*int*) – Seed for the random number generator. Defaults to 0.

**Attributes**

| | |
|---|---|
| **hyper-parameter_ranges** | { "step": Real(0.05, 0.25)} |
| **modifies_features** | True |
| **modifies_target** | False |
| **name** | RFE Selector with RF Regressor |
| **training_only** | False |

**Methods**

| | |
|---|---|
| *clone* | Constructs a new component with the same parameters and random state. |
| *default_parameters* | Returns the default parameters for this component. |
| *describe* | Describe a component and its parameters. |
| *fit* | Fits component to data. |
| *fit_transform* | Fit and transform data using the feature selector. |
| *get_names* | Get names of selected features. |
| *load* | Loads component at file path. |
| *needs_fitting* | Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances. |
| *parameters* | Returns the parameters which were used to initialize the component. |
| *save* | Saves component at file path. |
| *transform* | Transforms input data by selecting features. If the component_obj does not have a transform method, will raise an MethodPropertyNotFoundError exception. |
| *update_parameters* | Updates the parameter dictionary of the component. |

**clone**(*self*)

Constructs a new component with the same parameters and random state.

**Returns** A new instance of this component with identical parameters and random state.

**default_parameters**(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

**Returns** Default parameters for this component.

**Return type** dict

**describe**(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

**Parameters**

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description as dictionary in the format {"name": name, "parameters": parameters}

**Returns** Returns dictionary if return_dict is True, else None.

**Return type** None or dict

**fit**(*self*, *X*, *y=None*)

Fits component to data.

**Parameters**

- **X** (*pd.DataFrame*) – The input training data of shape [n_samples, n_features]
- **y** (*pd.Series, optional*) – The target training data of length [n_samples]

**Returns** self

**Raises** **MethodPropertyNotFoundError** – If component does not have a fit method or a component_obj that implements fit.

**fit_transform**(*self*, *X*, *y=None*)

Fit and transform data using the feature selector.

**Parameters**

- **X** (*pd.DataFrame*) – The input training data of shape [n_samples, n_features].
- **y** (*pd.Series, optional*) – The target training data of length [n_samples].

**Returns** Transformed data.

**Return type** pd.DataFrame

**get_names**(*self*)

Get names of selected features.

**Returns** List of the names of features selected.

**Return type** list[str]

**static load**(*file_path*)

Loads component at file path.

**Parameters** **file_path** (*str*) – Location to load file.

**Returns** ComponentBase object

**needs_fitting**(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need to be fit or whose fit methods do nothing.

**Returns** True.

**property parameters**(*self*)

Returns the parameters which were used to initialize the component.

**save**(*self*, *file_path*, *pickle_protocol=cloudpickle.DEFAULT_PROTOCOL*)

Saves component at file path.

**Parameters**

- **file_path** (*str*) – Location to save file.
- **pickle_protocol** (*int*) – The pickle data stream format.

**transform**(*self*, *X*, *y=None*)

Transforms input data by selecting features. If the component_obj does not have a transform method, will raise an MethodPropertyNotFoundError exception.

**Parameters**

- **X** (*pd.DataFrame*) – Data to transform.
- **y** (*pd.Series, optional*) – Target data. Ignored.

**Returns** Transformed X

**Return type** pd.DataFrame

**Raises** **MethodPropertyNotFoundError** – If feature selector does not have a transform method or a component_obj that implements transform

**update_parameters**(*self*, *update_dict*, *reset_fit=True*)

Updates the parameter dictionary of the component.

**Parameters**

- **update_dict** (*dict*) – A dict of parameters to update.
- **reset_fit** (*bool, optional*) – If True, will set *_is_fitted* to False.

class evalml.pipelines.components.transformers.**RFRegressorSelectFromModel**(*number_features=None*, *n_estimators=10*, *max_depth=None*, *percent_features=0.5*, *threshold='median'*, *n_jobs=- 1*, *random_seed=0*, ***kwargs*)

Selects top features based on importance weights using a Random Forest regressor.

**Parameters**

- **number_features** (*int*) – The maximum number of features to select. If both percent_features and number_features are specified, take the greater number of features. Defaults to 0.5.
- **n_estimators** (*int*) – The number of trees in the forest. Defaults to 10.
- **max_depth** (*int*) – Maximum tree depth for base learners. Defaults to None.
- **percent_features** (*float*) – Percentage of features to use. If both percent_features and number_features are specified, take the greater number of features. Defaults to 0.5.
- **threshold** (*string or float*) – The threshold value to use for feature selection. Features whose importance is greater or equal are kept while the others are discarded. If "median", then the threshold value is the median of the feature importances. A scaling factor (e.g., "1.25*mean") may also be used. Defaults to median.

- **n_jobs** (*int or None*) – Number of jobs to run in parallel. -1 uses all processes. Defaults to -1.
- **random_seed** (*int*) – Seed for the random number generator. Defaults to 0.

**Attributes**

| | |
|---|---|
| **hyper-parameter_ranges** | { "percent_features": Real(0.01, 1), "threshold": ["mean", "median"],} |
| **modifies_features** | True |
| **modifies_target** | False |
| **name** | RF Regressor Select From Model |
| **training_only** | False |

**Methods**

| | |
|---|---|
| *clone* | Constructs a new component with the same parameters and random state. |
| *default_parameters* | Returns the default parameters for this component. |
| *describe* | Describe a component and its parameters. |
| *fit* | Fits component to data. |
| *fit_transform* | Fit and transform data using the feature selector. |
| *get_names* | Get names of selected features. |
| *load* | Loads component at file path. |
| *needs_fitting* | Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances. |
| *parameters* | Returns the parameters which were used to initialize the component. |
| *save* | Saves component at file path. |
| *transform* | Transforms input data by selecting features. If the component_obj does not have a transform method, will raise an MethodPropertyNotFoundError exception. |
| *update_parameters* | Updates the parameter dictionary of the component. |

**clone**(*self*)

Constructs a new component with the same parameters and random state.

**Returns** A new instance of this component with identical parameters and random state.

**default_parameters**(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

**Returns** Default parameters for this component.

**Return type** dict

**describe**(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

**Parameters**

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description as dictionary in the format {"name": name, "parameters": parameters}

**Returns** Returns dictionary if return_dict is True, else None.

**Return type** None or dict

**fit**(*self*, *X*, *y=None*)

Fits component to data.

**Parameters**

- **X** (*pd.DataFrame*) – The input training data of shape [n_samples, n_features]
- **y** (*pd.Series, optional*) – The target training data of length [n_samples]

**Returns** self

**Raises** **MethodPropertyNotFoundError** – If component does not have a fit method or a component_obj that implements fit.

**fit_transform**(*self*, *X*, *y=None*)

Fit and transform data using the feature selector.

**Parameters**

- **X** (*pd.DataFrame*) – The input training data of shape [n_samples, n_features].
- **y** (*pd.Series, optional*) – The target training data of length [n_samples].

**Returns** Transformed data.

**Return type** pd.DataFrame

**get_names**(*self*)

Get names of selected features.

**Returns** List of the names of features selected.

**Return type** list[str]

**static load**(*file_path*)

Loads component at file path.

**Parameters** **file_path** (*str*) – Location to load file.

**Returns** ComponentBase object

**needs_fitting**(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need to be fit or whose fit methods do nothing.

**Returns** True.

**property parameters**(*self*)

Returns the parameters which were used to initialize the component.

**save**(*self*, *file_path*, *pickle_protocol=cloudpickle.DEFAULT_PROTOCOL*)

Saves component at file path.

**Parameters**

- **file_path** (*str*) – Location to save file.
- **pickle_protocol** (*int*) – The pickle data stream format.

**transform**(*self*, *X*, *y=None*)

Transforms input data by selecting features. If the component_obj does not have a transform method, will raise an MethodPropertyNotFoundError exception.

**Parameters**

- **X** (*pd.DataFrame*) – Data to transform.
- **y** (*pd.Series, optional*) – Target data. Ignored.

**Returns** Transformed X

**Return type** pd.DataFrame

**Raises** **MethodPropertyNotFoundError** – If feature selector does not have a transform method or a component_obj that implements transform

**update_parameters**(*self*, *update_dict*, *reset_fit=True*)

Updates the parameter dictionary of the component.

**Parameters**

- **update_dict** (*dict*) – A dict of parameters to update.
- **reset_fit** (*bool, optional*) – If True, will set *_is_fitted* to False.

**class** evalml.pipelines.components.transformers.**SelectByType**(*column_types=None*, *exclude=False*, *random_seed=0*, ***kwargs*)

Selects columns by specified Woodwork logical type or semantic tag in input data.

**Parameters**

- **column_types** (*string, ww.LogicalType, list(string), list(ww.LogicalType)*) – List of Woodwork types or tags, used to determine which columns to select or exclude.
- **exclude** (*bool*) – If true, exclude the column_types instead of including them. Defaults to False.
- **random_seed** (*int*) – Seed for the random number generator. Defaults to 0.

**Attributes**

| | |
|---|---|
| **hyper-parameter_ranges** | {} |
| **modifies_features** | True |
| **modifies_target** | False |
| **name** | Select Columns By Type Transformer |
| **needs_fitting** | False |
| **training_only** | False |

**Methods**

| | |
|---|---|
| *clone* | Constructs a new component with the same parameters and random state. |
| *default_parameters* | Returns the default parameters for this component. |
| *describe* | Describe a component and its parameters. |
| *fit* | Fits the transformer by checking if column names are present in the dataset. |
| *fit_transform* | Fits on X and transforms X. |
| *load* | Loads component at file path. |
| *parameters* | Returns the parameters which were used to initialize the component. |
| *save* | Saves component at file path. |
| *transform* | Transforms data X by selecting columns. |
| *update_parameters* | Updates the parameter dictionary of the component. |

**clone**(*self*)

Constructs a new component with the same parameters and random state.

**Returns** A new instance of this component with identical parameters and random state.

**default_parameters**(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

**Returns** Default parameters for this component.

**Return type** dict

**describe**(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

**Parameters**

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description as dictionary in the format {"name": name, "parameters": parameters}

**Returns** Returns dictionary if return_dict is True, else None.

**Return type** None or dict

**fit**(*self*, *X*, *y=None*)

Fits the transformer by checking if column names are present in the dataset.

**Parameters**

- **X** (*pd.DataFrame*) – Data to check.
- **y** (*pd.Series, ignored*) – Targets.

**Returns** self

**fit_transform**(*self*, *X*, *y=None*)

Fits on X and transforms X.

**Parameters**

- **X** (*pd.DataFrame*) – Data to fit and transform.

- **y** (*pd.Series*) – Target data.

**Returns** Transformed X.

**Return type** pd.DataFrame

**Raises** **MethodPropertyNotFoundError** – If transformer does not have a transform method or a component_obj that implements transform.

**static load**(*file_path*)

Loads component at file path.

**Parameters** **file_path** (*str*) – Location to load file.

**Returns** ComponentBase object

**property parameters**(*self*)

Returns the parameters which were used to initialize the component.

**save**(*self*, *file_path*, *pickle_protocol=cloudpickle.DEFAULT_PROTOCOL*)

Saves component at file path.

**Parameters**

- **file_path** (*str*) – Location to save file.
- **pickle_protocol** (*int*) – The pickle data stream format.

**transform**(*self*, *X*, *y=None*)

Transforms data X by selecting columns.

**Parameters**

- **X** (*pd.DataFrame*) – Data to transform.
- **y** (*pd.Series, optional*) – Targets.

**Returns** Transformed X.

**Return type** pd.DataFrame

**update_parameters**(*self*, *update_dict*, *reset_fit=True*)

Updates the parameter dictionary of the component.

**Parameters**

- **update_dict** (*dict*) – A dict of parameters to update.
- **reset_fit** (*bool, optional*) – If True, will set *_is_fitted* to False.

**class** evalml.pipelines.components.transformers.**SelectColumns**(*columns=None*, *random_seed=0*, ***kwargs*)

Selects specified columns in input data.

**Parameters**

- **columns** (*list(string)*) – List of column names, used to determine which columns to select. If columns are not present, they will not be selected.
- **random_seed** (*int*) – Seed for the random number generator. Defaults to 0.

**Attributes**

| | |
|---|---|
| **hyper-parameter_ranges** | {} |
| **modifies_features** | True |
| **modifies_target** | False |
| **name** | Select Columns Transformer |
| **needs_fitting** | False |
| **training_only** | False |

**Methods**

| | |
|---|---|
| *clone* | Constructs a new component with the same parameters and random state. |
| *default_parameters* | Returns the default parameters for this component. |
| *describe* | Describe a component and its parameters. |
| *fit* | Fits the transformer by checking if column names are present in the dataset. |
| *fit_transform* | Fits on X and transforms X. |
| *load* | Loads component at file path. |
| *parameters* | Returns the parameters which were used to initialize the component. |
| *save* | Saves component at file path. |
| *transform* | Transform data using fitted column selector component. |
| *update_parameters* | Updates the parameter dictionary of the component. |

**clone**(*self*)

Constructs a new component with the same parameters and random state.

**Returns** A new instance of this component with identical parameters and random state.

**default_parameters**(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

**Returns** Default parameters for this component.

**Return type** dict

**describe**(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

**Parameters**

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description as dictionary in the format {"name": name, "parameters": parameters}

**Returns** Returns dictionary if return_dict is True, else None.

**Return type** None or dict

**fit**(*self*, *X*, *y=None*)

Fits the transformer by checking if column names are present in the dataset.

**Parameters**

- **X** (*pd.DataFrame*) – Data to check.
- **y** (*pd.Series, optional*) – Targets.

**Returns** self

**fit_transform**(*self*, *X*, *y=None*)

Fits on X and transforms X.

**Parameters**

- **X** (*pd.DataFrame*) – Data to fit and transform.
- **y** (*pd.Series*) – Target data.

**Returns** Transformed X.

**Return type** pd.DataFrame

**Raises** **MethodPropertyNotFoundError** – If transformer does not have a transform method or a component_obj that implements transform.

**static load**(*file_path*)

Loads component at file path.

**Parameters** **file_path** (*str*) – Location to load file.

**Returns** ComponentBase object

**property parameters**(*self*)

Returns the parameters which were used to initialize the component.

**save**(*self*, *file_path*, *pickle_protocol=cloudpickle.DEFAULT_PROTOCOL*)

Saves component at file path.

**Parameters**

- **file_path** (*str*) – Location to save file.
- **pickle_protocol** (*int*) – The pickle data stream format.

**transform**(*self*, *X*, *y=None*)

Transform data using fitted column selector component.

**Parameters**

- **X** (*pd.DataFrame*) – The input training data of shape [n_samples, n_features].
- **y** (*pd.Series, optional*) – The target training data of length [n_samples].

**Returns** Transformed data.

**Return type** pd.DataFrame

**update_parameters**(*self*, *update_dict*, *reset_fit=True*)

Updates the parameter dictionary of the component.

**Parameters**

- **update_dict** (*dict*) – A dict of parameters to update.
- **reset_fit** (*bool, optional*) – If True, will set *_is_fitted* to False.

**class** evalml.pipelines.components.transformers.**SimpleImputer**(*impute_strategy='most_frequent', fill_value=None, random_seed=0, **kwargs*)

Imputes missing data according to a specified imputation strategy. Natural language columns are ignored.

**Parameters**

- **impute_strategy** (*string*) – Impute strategy to use. Valid values include "mean", "median", "most_frequent", "constant" for numerical data, and "most_frequent", "constant" for object data types.
- **fill_value** (*string*) – When impute_strategy == "constant", fill_value is used to replace missing data. Defaults to 0 when imputing numerical data and "missing_value" for strings or object data types.
- **random_seed** (*int*) – Seed for the random number generator. Defaults to 0.

**Attributes**

| | |
|---|---|
| **hyper-parameter_ranges** | { "impute_strategy": ["mean", "median", "most_frequent"]} |
| **modifies_features** | True |
| **modifies_target** | False |
| **name** | Simple Imputer |
| **training_only** | False |

**Methods**

| | |
|---|---|
| *clone* | Constructs a new component with the same parameters and random state. |
| *default_parameters* | Returns the default parameters for this component. |
| *describe* | Describe a component and its parameters. |
| *fit* | Fits imputer to data. 'None' values are converted to np.nan before imputation and are treated as the same. |
| *fit_transform* | Fits on X and transforms X. |
| *load* | Loads component at file path. |
| *needs_fitting* | Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances. |
| *parameters* | Returns the parameters which were used to initialize the component. |
| *save* | Saves component at file path. |
| *transform* | Transforms input by imputing missing values. 'None' and np.nan values are treated as the same. |
| *update_parameters* | Updates the parameter dictionary of the component. |

**clone**(*self*)

Constructs a new component with the same parameters and random state.

**Returns** A new instance of this component with identical parameters and random state.

**default_parameters**(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

**Returns** Default parameters for this component.

**Return type** dict

**describe**(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

**Parameters**

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description as dictionary in the format {"name": name, "parameters": parameters}

**Returns** Returns dictionary if return_dict is True, else None.

**Return type** None or dict

**fit**(*self*, *X*, *y=None*)

Fits imputer to data. 'None' values are converted to np.nan before imputation and are treated as the same.

**Parameters**

- **X** (*pd.DataFrame or np.ndarray*) – the input training data of shape [n_samples, n_features]
- **y** (*pd.Series, optional*) – the target training data of length [n_samples]

**Returns** self

**Raises** **ValueError** – if the SimpleImputer receives a dataframe with both Boolean and Categorical data.

**fit_transform**(*self*, *X*, *y=None*)

Fits on X and transforms X.

**Parameters**

- **X** (*pd.DataFrame*) – Data to fit and transform
- **y** (*pd.Series, optional*) – Target data.

**Returns** Transformed X

**Return type** pd.DataFrame

**static load**(*file_path*)

Loads component at file path.

**Parameters** **file_path** (*str*) – Location to load file.

**Returns** ComponentBase object

**needs_fitting**(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need to be fit or whose fit methods do nothing.

**Returns** True.

**property parameters**(*self*)

Returns the parameters which were used to initialize the component.

**save**(*self*, *file_path*, *pickle_protocol=cloudpickle.DEFAULT_PROTOCOL*)

Saves component at file path.

**Parameters**

- **file_path** (*str*) – Location to save file.
- **pickle_protocol** (*int*) – The pickle data stream format.

**transform**(*self*, *X*, *y=None*)

Transforms input by imputing missing values. 'None' and np.nan values are treated as the same.

**Parameters**

- **X** (*pd.DataFrame*) – Data to transform.
- **y** (*pd.Series, optional*) – Ignored.

**Returns** Transformed X

**Return type** pd.DataFrame

**update_parameters**(*self*, *update_dict*, *reset_fit=True*)

Updates the parameter dictionary of the component.

**Parameters**

- **update_dict** (*dict*) – A dict of parameters to update.
- **reset_fit** (*bool, optional*) – If True, will set *_is_fitted* to False.

**class** evalml.pipelines.components.transformers.**StandardScaler**(*random_seed=0*, ***kwargs*)

A transformer that standardizes input features by removing the mean and scaling to unit variance.

**Parameters** **random_seed** (*int*) – Seed for the random number generator. Defaults to 0.

**Attributes**

| | |
|---|---|
| **hyper-parameter_ranges** | {} |
| **modifies_features** | True |
| **modifies_target** | False |
| **name** | Standard Scaler |
| **training_only** | False |

**Methods**

| | |
|---|---|
| *clone* | Constructs a new component with the same parameters and random state. |
| *default_parameters* | Returns the default parameters for this component. |
| *describe* | Describe a component and its parameters. |
| *fit* | Fits the standard scalar on the given data. |
| *fit_transform* | Fit and transform data using the standard scaler component. |
| *load* | Loads component at file path. |
| *needs_fitting* | Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances. |
| *parameters* | Returns the parameters which were used to initialize the component. |
| *save* | Saves component at file path. |
| *transform* | Transform data using the fitted standard scaler. |
| *update_parameters* | Updates the parameter dictionary of the component. |

**clone**(*self*)

Constructs a new component with the same parameters and random state.

**Returns** A new instance of this component with identical parameters and random state.

**default_parameters**(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

**Returns** Default parameters for this component.

**Return type** dict

**describe**(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

**Parameters**

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description as dictionary in the format {"name": name, "parameters": parameters}

**Returns** Returns dictionary if return_dict is True, else None.

**Return type** None or dict

**fit**(*self*, *X*, *y=None*)

Fits the standard scalar on the given data.

**Parameters**

- **X** (*pd.DataFrame*) – The input training data of shape [n_samples, n_features].
- **y** (*pd.Series, optional*) – The target training data of length [n_samples].

**Returns** self

**fit_transform**(*self*, *X*, *y=None*)

Fit and transform data using the standard scaler component.

**Parameters**

- **X** (*pd.DataFrame*) – The input training data of shape [n_samples, n_features].
- **y** (*pd.Series, optional*) – The target training data of length [n_samples].

**Returns** Transformed data.

**Return type** pd.DataFrame

**static load**(*file_path*)

Loads component at file path.

**Parameters** **file_path** (*str*) – Location to load file.

**Returns** ComponentBase object

**needs_fitting**(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need to be fit or whose fit methods do nothing.

**Returns** True.

**property parameters**(*self*)

Returns the parameters which were used to initialize the component.

**save**(*self*, *file_path*, *pickle_protocol=cloudpickle.DEFAULT_PROTOCOL*)

Saves component at file path.

**Parameters**

- **file_path** (*str*) – Location to save file.
- **pickle_protocol** (*int*) – The pickle data stream format.

**transform**(*self*, *X*, *y=None*)

Transform data using the fitted standard scaler.

**Parameters**

- **X** (*pd.DataFrame*) – The input training data of shape [n_samples, n_features].
- **y** (*pd.Series, optional*) – The target training data of length [n_samples].

**Returns** Transformed data.

**Return type** pd.DataFrame

**update_parameters**(*self*, *update_dict*, *reset_fit=True*)

Updates the parameter dictionary of the component.

**Parameters**

- **update_dict** (*dict*) – A dict of parameters to update.
- **reset_fit** (*bool, optional*) – If True, will set *_is_fitted* to False.

**class** evalml.pipelines.components.transformers.**STLDecomposer**(*time_index: str = None*, *degree: int = 1*, *period: int = None*, *seasonal_smoother: int = 7*, *random_seed: int = 0*, ***kwargs*)

Removes trends and seasonality from time series using the STL algorithm.

https://www.statsmodels.org/dev/generated/statsmodels.tsa.seasonal.STL.html

**Parameters**

- **time_index** (*str*) – Specifies the name of the column in X that provides the datetime objects. Defaults to None.
- **degree** (*int*) – Not currently used. STL 3x "degree-like" values. None are able to be set at this time. Defaults to 1.
- **period** (*int*) – The number of entries in the time series data that corresponds to one period of a cyclic signal. For instance, if data is known to possess a weekly seasonal signal, and if the data is daily data, the period should likely be 7. For daily data with a yearly seasonal signal, the period should likely be 365. If None, statsmodels will infer the period based on the frequency. Defaults to None.
- **seasonal_smoother** (*int*) – The length of the seasonal smoother used by the underlying STL algorithm. For compatibility, must be odd. If an even number is provided, the next, highest odd number will be used. Defaults to 7.
- **random_seed** (*int*) – Seed for the random number generator. Defaults to 0.

**Attributes**

| | |
|---|---|
| **hyperparameter_ranges** | None |
| **invalid_frequencies** | [] |
| **modifies_features** | False |
| **modifies_target** | True |
| **name** | STL Decomposer |
| **needs_fitting** | True |
| **training_only** | False |

**Methods**

| | |
|---|---|
| *clone* | Constructs a new component with the same parameters and random state. |
| *default_parameters* | Returns the default parameters for this component. |
| *describe* | Describe a component and its parameters. |
| *determine_periodicity* | Function that uses autocorrelative methods to determine the likely most signficant period of the seasonal signal. |
| *fit* | Fits the STLDecomposer and determine the seasonal signal. |
| *fit_transform* | Removes fitted trend and seasonality from target variable. |
| *get_trend_dataframe* | Return a list of dataframes with 4 columns: signal, trend, seasonality, residual. |
| *get_trend_prediction_intervals* | Calculate the prediction intervals for the trend data. |
| *inverse_transform* | Adds back fitted trend and seasonality to target variable. |
| *is_freq_valid* | Determines if the given string represents a valid frequency for this decomposer. |
| *load* | Loads component at file path. |
| *parameters* | Returns the parameters which were used to initialize the component. |
| *plot_decomposition* | Plots the decomposition of the target signal. |
| *save* | Saves component at file path. |
| *set_period* | Function to set the component's seasonal period based on the target's seasonality. |
| *transform* | Transforms the target data by removing the STL trend and seasonality. |
| *update_parameters* | Updates the parameter dictionary of the component. |

**clone**(*self*)

Constructs a new component with the same parameters and random state.

**Returns** A new instance of this component with identical parameters and random state.

**default_parameters**(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

**Returns** Default parameters for this component.

**Return type** dict

**describe**(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

**Parameters**

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description as dictionary in the format {"name": name, "parameters": parameters}

**Returns** Returns dictionary if return_dict is True, else None.

**Return type** None or dict

**classmethod determine_periodicity**(*cls*, *X: pandas.DataFrame*, *y: pandas.Series*, *acf_threshold: float = 0.01*, *rel_max_order: int = 5*)

Function that uses autocorrelative methods to determine the likely most signficant period of the seasonal signal.

**Parameters**

- **X** (*pandas.DataFrame*) – The feature data of the time series problem.
- **y** (*pandas.Series*) – The target data of a time series problem.
- **acf_threshold** (*float*) – The threshold for the autocorrelation function to determine the period. Any values below the threshold are considered to be 0 and will not be considered for the period. Defaults to 0.01.
- **rel_max_order** (*int*) – The order of the relative maximum to determine the period. Defaults to 5.

**Returns**

**The integer number of entries in time series data over which the seasonal part of the target data** repeats. If the time series data is in days, then this is the number of days that it takes the target's seasonal signal to repeat. Note: the target data can contain multiple seasonal signals. This function will only return the stronger. E.g. if the target has both weekly and yearly seasonality, the function may return either "7" or "365", depending on which seasonality is more strongly autocorrelated. If no period is detected, returns None.

**Return type** int

**fit**(*self*, *X: pandas.DataFrame*, *y: pandas.Series = None*) → *STLDecomposer*

Fits the STLDecomposer and determine the seasonal signal.

Instantiates a statsmodels STL decompose object with the component's stored parameters and fits it. Since the statsmodels object does not fit the sklearn api, it is not saved during __init__() in _component_obj and will be re-instantiated each time fit is called.

To emulate the sklearn API, when the STL decomposer is fit, the full seasonal component, a single period sample of the seasonal component, the full trend-cycle component and the residual are saved.

y(t) = S(t) + T(t) + R(t)

**Parameters**

- **X** (*pd.DataFrame, optional*) – Conditionally used to build datetime index.
- **y** (*pd.Series*) – Target variable to detrend and deseasonalize.

**Returns** self

**Raises**

- **ValueError** – If y is None.
- **ValueError** – If target data doesn't have DatetimeIndex AND no Datetime features in features data

**fit_transform**(*self*, *X: pandas.DataFrame*, *y: pandas.Series = None*) → tuple[pandas.DataFrame, pandas.Series]

Removes fitted trend and seasonality from target variable.

**Parameters**

- **X** (*pd.DataFrame, optional*) – Ignored.
- **y** (*pd.Series*) – Target variable to detrend and deseasonalize.

**Returns**

**The first element are the input features returned without modification.** The second element is the target variable y with the fitted trend removed.

**Return type** tuple of pd.DataFrame, pd.Series

`get_trend_dataframe`(*self*, *X*, *y*)

Return a list of dataframes with 4 columns: signal, trend, seasonality, residual.

**Parameters**

- **X** (*pd.DataFrame*) – Input data with time series data in index.
- **y** (*pd.Series or pd.DataFrame*) – Target variable data provided as a Series for univariate problems or a DataFrame for multivariate problems.

**Returns**

**Each DataFrame contains the columns "signal", "trend", "seasonality" and "residual,"** with the latter 3 column values being the decomposed elements of the target data. The "signal" column is simply the input target signal but reindexed with a datetime index to match the input features.

**Return type** list of pd.DataFrame

**Raises**

- **TypeError** – If X does not have time-series data in the index.
- **ValueError** – If time series index of X does not have an inferred frequency.
- **ValueError** – If the forecaster associated with the detrender has not been fit yet.
- **TypeError** – If y is not provided as a pandas Series or DataFrame.

`get_trend_prediction_intervals`(*self*, *y*, *coverage=None*)

Calculate the prediction intervals for the trend data.

**Parameters**

- **y** (*pd.Series*) – Target data.
- **coverage** (*list[float]*) – A list of floats between the values 0 and 1 that the upper and lower bounds of the prediction interval should be calculated for.

**Returns** Prediction intervals, keys are in the format {coverage}_lower or {coverage}_upper.

**Return type** dict of pd.Series

`inverse_transform`(*self*, *y_t: pandas.Series*) → tuple[pandas.DataFrame, pandas.Series]

Adds back fitted trend and seasonality to target variable.

The STL trend is projected to cover the entire requested target range, then added back into the signal. Then, the seasonality is projected forward to and added back into the signal.

**Parameters** **y_t** (*pd.Series*) – Target variable.

**Returns**

**The first element are the input features returned without modification.** The second element is the target variable y with the trend and seasonality added back in.

**Return type** tuple of pd.DataFrame, pd.Series

**Raises** **ValueError** – If y is None.

**classmethod is_freq_valid**(*cls*, *freq: str*)

Determines if the given string represents a valid frequency for this decomposer.

**Parameters** **freq** (*str*) – A frequency to validate. See the pandas docs at https://pandas.pydata.org/pandas-docs/stable/user_guide/timeseries.html#offset-aliases for options.

**Returns** boolean representing whether the frequency is valid or not.

**static load**(*file_path*)

Loads component at file path.

**Parameters** **file_path** (*str*) – Location to load file.

**Returns** ComponentBase object

**property parameters**(*self*)

Returns the parameters which were used to initialize the component.

**plot_decomposition**(*self*, *X: pandas.DataFrame*, *y: pandas.Series*, *show: bool = False*) → tuple[matplotlib.pyplot.Figure, list]

Plots the decomposition of the target signal.

**Parameters**

- **X** (*pd.DataFrame*) – Input data with time series data in index.
- **y** (*pd.Series or pd.DataFrame*) – Target variable data provided as a Series for univariate problems or a DataFrame for multivariate problems.
- **show** (*bool*) – Whether to display the plot or not. Defaults to False.

**Returns**

**The figure and axes that have the decompositions** plotted on them

**Return type** matplotlib.pyplot.Figure, list[matplotlib.pyplot.Axes]

**save**(*self*, *file_path*, *pickle_protocol=cloudpickle.DEFAULT_PROTOCOL*)

Saves component at file path.

**Parameters**

- **file_path** (*str*) – Location to save file.
- **pickle_protocol** (*int*) – The pickle data stream format.

**set_period**(*self*, *X: pandas.DataFrame*, *y: pandas.Series*, *acf_threshold: float = 0.01*, *rel_max_order: int = 5*)

Function to set the component's seasonal period based on the target's seasonality.

**Parameters**

- **X** (*pandas.DataFrame*) – The feature data of the time series problem.
- **y** (*pandas.Series*) – The target data of a time series problem.
- **acf_threshold** (*float*) – The threshold for the autocorrelation function to determine the period. Any values below the threshold are considered to be 0 and will not be considered for the period. Defaults to 0.01.
- **rel_max_order** (*int*) – The order of the relative maximum to determine the period. Defaults to 5.

**transform**(*self*, *X: pandas.DataFrame*, *y: pandas.Series = None*) → tuple[pandas.DataFrame, pandas.Series]

Transforms the target data by removing the STL trend and seasonality.

Uses an ARIMA model to project forward the addititve trend and removes it. Then, utilizes the first period's worth of seasonal data determined in the .fit() function to extrapolate the seasonal signal of the data to be transformed. This seasonal signal is also assumed to be additive and is removed.

**Parameters**

- **X** (*pd.DataFrame, optional*) – Conditionally used to build datetime index.
- **y** (*pd.Series*) – Target variable to detrend and deseasonalize.

**Returns**

**The input features are returned without modification. The target** variable y is detrended and deseasonalized.

**Return type** tuple of pd.DataFrame, pd.Series

**Raises** **ValueError** – If target data doesn't have DatetimeIndex AND no Datetime features in features data

**update_parameters**(*self*, *update_dict*, *reset_fit=True*)

Updates the parameter dictionary of the component.

**Parameters**

- **update_dict** (*dict*) – A dict of parameters to update.
- **reset_fit** (*bool, optional*) – If True, will set *_is_fitted* to False.

**class** evalml.pipelines.components.transformers.**TargetEncoder**(*cols=None*, *smoothing=1*, *handle_unknown='value'*, *handle_missing='value'*, *random_seed=0*, ***kwargs*)

A transformer that encodes categorical features into target encodings.

**Parameters**

- **cols** (*list*) – Columns to encode. If None, all string columns will be encoded, otherwise only the columns provided will be encoded. Defaults to None
- **smoothing** (*float*) – The smoothing factor to apply. The larger this value is, the more influence the expected target value has on the resulting target encodings. Must be strictly larger than 0. Defaults to 1.0
- **handle_unknown** (*string*) – Determines how to handle unknown categories for a feature encountered. Options are 'value', 'error', nd 'return_nan'. Defaults to 'value', which replaces with the target mean
- **handle_missing** (*string*) – Determines how to handle missing values encountered during *fit* or *transform*. Options are 'value', 'error', and 'return_nan'. Defaults to 'value', which replaces with the target mean
- **random_seed** (*int*) – Seed for the random number generator. Defaults to 0.

**Attributes**

| | |
|---|---|
| **hyper-parameter_ranges** | {} |
| **modifies_features** | True |
| **modifies_target** | False |
| **name** | Target Encoder |
| **training_only** | False |

**Methods**

| | |
|---|---|
| *clone* | Constructs a new component with the same parameters and random state. |
| *default_parameters* | Returns the default parameters for this component. |
| *describe* | Describe a component and its parameters. |
| *fit* | Fits the target encoder. |
| *fit_transform* | Fit and transform data using the target encoder. |
| *get_feature_names* | Return feature names for the input features after fitting. |
| *load* | Loads component at file path. |
| *needs_fitting* | Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances. |
| *parameters* | Returns the parameters which were used to initialize the component. |
| *save* | Saves component at file path. |
| *transform* | Transform data using the fitted target encoder. |
| *update_parameters* | Updates the parameter dictionary of the component. |

**clone**(*self*)

Constructs a new component with the same parameters and random state.

**Returns** A new instance of this component with identical parameters and random state.

**default_parameters**(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

**Returns** Default parameters for this component.

**Return type** dict

**describe**(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

**Parameters**

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description as dictionary in the format {"name": name, "parameters": parameters}

**Returns** Returns dictionary if return_dict is True, else None.

**Return type** None or dict

**fit**(*self*, *X*, *y*)

Fits the target encoder.

**Parameters**

- **X** (*pd.DataFrame*) – The input training data of shape [n_samples, n_features].
- **y** (*pd.Series, optional*) – The target training data of length [n_samples].

**Returns** self

**fit_transform**(*self*, *X*, *y*)

Fit and transform data using the target encoder.

**Parameters**

- **X** (*pd.DataFrame*) – The input training data of shape [n_samples, n_features].
- **y** (*pd.Series, optional*) – The target training data of length [n_samples].

**Returns** Transformed data.

**Return type** pd.DataFrame

**get_feature_names**(*self*)

Return feature names for the input features after fitting.

**Returns** The feature names after encoding.

**Return type** np.array

**static load**(*file_path*)

Loads component at file path.

**Parameters** **file_path** (*str*) – Location to load file.

**Returns** ComponentBase object

**needs_fitting**(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need to be fit or whose fit methods do nothing.

**Returns** True.

**property parameters**(*self*)

Returns the parameters which were used to initialize the component.

**save**(*self*, *file_path*, *pickle_protocol=cloudpickle.DEFAULT_PROTOCOL*)

Saves component at file path.

**Parameters**

- **file_path** (*str*) – Location to save file.
- **pickle_protocol** (*int*) – The pickle data stream format.

**transform**(*self*, *X*, *y=None*)

Transform data using the fitted target encoder.

**Parameters**

- **X** (*pd.DataFrame*) – The input training data of shape [n_samples, n_features].

- **y** (*pd.Series, optional*) – The target training data of length [n_samples].

**Returns** Transformed data.

**Return type** pd.DataFrame

**update_parameters**(*self*, *update_dict*, *reset_fit=True*)

Updates the parameter dictionary of the component.

**Parameters**

- **update_dict** (*dict*) – A dict of parameters to update.
- **reset_fit** (*bool, optional*) – If True, will set *_is_fitted* to False.

**class** evalml.pipelines.components.transformers.**TargetImputer**(*impute_strategy='most_frequent', fill_value=None, random_seed=0, **kwargs*)

Imputes missing target data according to a specified imputation strategy.

**Parameters**

- **impute_strategy** (*string*) – Impute strategy to use. Valid values include "mean", "median", "most_frequent", "constant" for numerical data, and "most_frequent", "constant" for object data types. Defaults to "most_frequent".
- **fill_value** (*string*) – When impute_strategy == "constant", fill_value is used to replace missing data. Defaults to None which uses 0 when imputing numerical data and "missing_value" for strings or object data types.
- **random_seed** (*int*) – Seed for the random number generator. Defaults to 0.

**Attributes**

| | |
|---|---|
| **hyperparameter_ranges** | { "impute_strategy": ["mean", "median", "most_frequent"]} |
| **modifies_features** | False |
| **modifies_target** | True |
| **name** | Target Imputer |
| **training_only** | False |

**Methods**

| | |
|---|---|
| *clone* | Constructs a new component with the same parameters and random state. |
| *default_parameters* | Returns the default parameters for this component. |
| *describe* | Describe a component and its parameters. |
| *fit* | Fits imputer to target data. 'None' values are converted to np.nan before imputation and are treated as the same. |
| *fit_transform* | Fits on and transforms the input target data. |
| *load* | Loads component at file path. |
| *needs_fitting* | Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances. |
| *parameters* | Returns the parameters which were used to initialize the component. |
| *save* | Saves component at file path. |
| *transform* | Transforms input target data by imputing missing values. 'None' and np.nan values are treated as the same. |
| *update_parameters* | Updates the parameter dictionary of the component. |

**clone**(*self*)

Constructs a new component with the same parameters and random state.

**Returns** A new instance of this component with identical parameters and random state.

**default_parameters**(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

**Returns** Default parameters for this component.

**Return type** dict

**describe**(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

**Parameters**

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description as dictionary in the format {"name": name, "parameters": parameters}

**Returns** Returns dictionary if return_dict is True, else None.

**Return type** None or dict

**fit**(*self*, *X*, *y*)

Fits imputer to target data. 'None' values are converted to np.nan before imputation and are treated as the same.

**Parameters**

- **X** (*pd.DataFrame or np.ndarray*) – The input training data of shape [n_samples, n_features]. Ignored.
- **y** (*pd.Series, optional*) – The target training data of length [n_samples].

**Returns** self

**Raises** `TypeError` – If target is filled with all null values.

`fit_transform`(*self*, *X*, *y*)

Fits on and transforms the input target data.

**Parameters**

- `X` (*pd.DataFrame*) – Features. Ignored.
- `y` (*pd.Series*) – Target data to impute.

**Returns** The original X, transformed y

**Return type** (pd.DataFrame, pd.Series)

`static load`(*file_path*)

Loads component at file path.

**Parameters** `file_path` (*str*) – Location to load file.

**Returns** ComponentBase object

`needs_fitting`(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need to be fit or whose fit methods do nothing.

**Returns** True.

`property parameters`(*self*)

Returns the parameters which were used to initialize the component.

`save`(*self*, *file_path*, *pickle_protocol=cloudpickle.DEFAULT_PROTOCOL*)

Saves component at file path.

**Parameters**

- `file_path` (*str*) – Location to save file.
- `pickle_protocol` (*int*) – The pickle data stream format.

`transform`(*self*, *X*, *y*)

Transforms input target data by imputing missing values. 'None' and np.nan values are treated as the same.

**Parameters**

- `X` (*pd.DataFrame*) – Features. Ignored.
- `y` (*pd.Series*) – Target data to impute.

**Returns** The original X, transformed y

**Return type** (pd.DataFrame, pd.Series)

`update_parameters`(*self*, *update_dict*, *reset_fit=True*)

Updates the parameter dictionary of the component.

**Parameters**

- `update_dict` (*dict*) – A dict of parameters to update.
- `reset_fit` (*bool, optional*) – If True, will set *_is_fitted* to False.

**class** evalml.pipelines.components.transformers.**TimeSeriesFeaturizer**(*time_index=None*, *max_delay=2*, *gap=0*, *forecast_horizon=1*, *conf_level=0.05*, *rolling_window_size=0.25*, *delay_features=True*, *delay_target=True*, *random_seed=0*, ***kwargs*)

Transformer that delays input features and target variable for time series problems.

This component uses an algorithm based on the autocorrelation values of the target variable to determine which lags to select from the set of all possible lags.

The algorithm is based on the idea that the local maxima of the autocorrelation function indicate the lags that have the most impact on the present time.

The algorithm computes the autocorrelation values and finds the local maxima, called "peaks", that are significant at the given conf_level. Since lags in the range [0, 10] tend to be predictive but not local maxima, the union of the peaks is taken with the significant lags in the range [0, 10]. At the end, only selected lags in the range [0, max_delay] are used.

Parametrizing the algorithm by conf_level lets the AutoMLAlgorithm tune the set of lags chosen so that the chances of finding a good set of lags is higher.

Using conf_level value of 1 selects all possible lags.

**Parameters**

- **time_index** (*str*) – Name of the column containing the datetime information used to order the data. Ignored.
- **max_delay** (*int*) – Maximum number of time units to delay each feature. Defaults to 2.
- **forecast_horizon** (*int*) – The number of time periods the pipeline is expected to forecast.
- **conf_level** (*float*) – Float in range (0, 1] that determines the confidence interval size used to select which lags to compute from the set of [1, max_delay]. A delay of 1 will always be computed. If 1, selects all possible lags in the set of [1, max_delay], inclusive.
- **rolling_window_size** (*float*) – Float in range (0, 1] that determines the size of the window used for rolling features. Size is computed as rolling_window_size * max_delay.
- **delay_features** (*bool*) – Whether to delay the input features. Defaults to True.
- **delay_target** (*bool*) – Whether to delay the target. Defaults to True.
- **gap** (*int*) – The number of time units between when the features are collected and when the target is collected. For example, if you are predicting the next time step's target, gap=1. This is only needed because when gap=0, we need to be sure to start the lagging of the target variable at 1. Defaults to 1.
- **random_seed** (*int*) – Seed for the random number generator. This transformer performs the same regardless of the random seed provided.

**Attributes**

| | |
|---|---|
| **hyper-parameter_ranges** | Real(0.001, 1.0), "rolling_window_size": Real(0.001, 1.0)}:type: {"conf_level" |
| **modifies_features** | True |
| **modifies_target** | False |
| **name** | Time Series Featurizer |
| **needs_fitting** | True |
| **target_colname_prefix** | target_delay_{} |
| **training_only** | False |

**Methods**

| | |
|---|---|
| *clone* | Constructs a new component with the same parameters and random state. |
| *default_parameters* | Returns the default parameters for this component. |
| *describe* | Describe a component and its parameters. |
| *fit* | Fits the DelayFeatureTransformer. |
| *fit_transform* | Fit the component and transform the input data. |
| *load* | Loads component at file path. |
| *parameters* | Returns the parameters which were used to initialize the component. |
| *save* | Saves component at file path. |
| *transform* | Computes the delayed values and rolling means for X and y. |
| *update_parameters* | Updates the parameter dictionary of the component. |

**clone**(*self*)

Constructs a new component with the same parameters and random state.

**Returns** A new instance of this component with identical parameters and random state.

**default_parameters**(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

**Returns** Default parameters for this component.

**Return type** dict

**describe**(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

**Parameters**

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description as dictionary in the format {"name": name, "parameters": parameters}

**Returns** Returns dictionary if return_dict is True, else None.

**Return type** None or dict

**fit**(*self*, *X*, *y=None*)

Fits the DelayFeatureTransformer.

**Parameters**

- **X** (*pd.DataFrame or np.ndarray*) – The input training data of shape [n_samples, n_features]
- **y** (*pd.Series, optional*) – The target training data of length [n_samples]

**Returns** self

**Raises** **ValueError** – if self.time_index is None

**fit_transform**(*self*, *X*, *y=None*)

Fit the component and transform the input data.

**Parameters**

- **X** (*pd.DataFrame*) – Data to transform.
- **y** (*pd.Series, or None*) – Target.

**Returns** Transformed X.

**Return type** pd.DataFrame

**static load**(*file_path*)

Loads component at file path.

**Parameters** **file_path** (*str*) – Location to load file.

**Returns** ComponentBase object

**property parameters**(*self*)

Returns the parameters which were used to initialize the component.

**save**(*self*, *file_path*, *pickle_protocol=cloudpickle.DEFAULT_PROTOCOL*)

Saves component at file path.

**Parameters**

- **file_path** (*str*) – Location to save file.
- **pickle_protocol** (*int*) – The pickle data stream format.

**transform**(*self*, *X*, *y=None*)

Computes the delayed values and rolling means for X and y.

The chosen delays are determined by the autocorrelation function of the target variable. See the class docstring for more information on how they are chosen. If y is None, all possible lags are chosen.

If y is not None, it will also compute the delayed values for the target variable.

The rolling means for all numeric features in X and y, if y is numeric, are also returned.

**Parameters**

- **X** (*pd.DataFrame or None*) – Data to transform. None is expected when only the target variable is being used.
- **y** (*pd.Series, or None*) – Target.

**Returns** Transformed X. No original features are returned.

**Return type** pd.DataFrame

**update_parameters**(*self*, *update_dict*, *reset_fit=True*)

Updates the parameter dictionary of the component.

**Parameters**

- **update_dict** (*dict*) – A dict of parameters to update.
- **reset_fit** (*bool, optional*) – If True, will set *_is_fitted* to False.

**class** evalml.pipelines.components.transformers.**TimeSeriesImputer**(*categorical_impute_strategy='forwards_fill', numeric_impute_strategy='interpolate', target_impute_strategy='forwards_fill', random_seed=0, **kwargs*)

Imputes missing data according to a specified timeseries-specific imputation strategy.

This Transformer should be used after the *TimeSeriesRegularizer* in order to impute the missing values that were added to X and y (if passed).

**Parameters**

- **categorical_impute_strategy** (*string*) – Impute strategy to use for string, object, boolean, categorical dtypes. Valid values include "backwards_fill" and "forwards_fill". Defaults to "forwards_fill".
- **numeric_impute_strategy** (*string*) – Impute strategy to use for numeric columns. Valid values include "backwards_fill", "forwards_fill", and "interpolate". Defaults to "interpolate".
- **target_impute_strategy** (*string*) – Impute strategy to use for the target column. Valid values include "backwards_fill", "forwards_fill", and "interpolate". Defaults to "forwards_fill".
- **random_seed** (*int*) – Seed for the random number generator. Defaults to 0.

**Raises** **ValueError** – If categorical_impute_strategy, numeric_impute_strategy, or target_impute_strategy is not one of the valid values.

**Attributes**

| | |
|---|---|
| **hyperparameter_ranges** | { "categorical_impute_strategy": ["backwards_fill", "forwards_fill"], "numeric_impute_strategy": ["backwards_fill", "forwards_fill", "interpolate"], "target_impute_strategy": ["backwards_fill", "forwards_fill", "interpolate"],} |
| **modifies_features** | True |
| **modifies_target** | True |
| **name** | Time Series Imputer |
| **training_only** | True |

**Methods**

| | |
|---|---|
| *clone* | Constructs a new component with the same parameters and random state. |
| *default_parameters* | Returns the default parameters for this component. |
| *describe* | Describe a component and its parameters. |
| *fit* | Fits imputer to data. |
| *fit_transform* | Fits on X and transforms X. |
| *load* | Loads component at file path. |
| *needs_fitting* | Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances. |
| *parameters* | Returns the parameters which were used to initialize the component. |
| *save* | Saves component at file path. |
| *transform* | Transforms data X by imputing missing values using specified timeseries-specific strategies. 'None' values are converted to np.nan before imputation and are treated as the same. |
| *update_parameters* | Updates the parameter dictionary of the component. |

**clone**(*self*)

Constructs a new component with the same parameters and random state.

**Returns** A new instance of this component with identical parameters and random state.

**default_parameters**(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

**Returns** Default parameters for this component.

**Return type** dict

**describe**(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

**Parameters**

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description as dictionary in the format {"name": name, "parameters": parameters}

**Returns** Returns dictionary if return_dict is True, else None.

**Return type** None or dict

**fit**(*self*, *X*, *y=None*)

Fits imputer to data.

'None' values are converted to np.nan before imputation and are treated as the same. If a value is missing at the beginning or end of a column, that value will be imputed using backwards fill or forwards fill as necessary, respectively.

**Parameters**

- **X** (*pd.DataFrame, np.ndarray*) – The input training data of shape [n_samples, n_features]
- **y** (*pd.Series, optional*) – The target training data of length [n_samples]

**Returns** self

`fit_transform`(*self*, *X*, *y=None*)

Fits on X and transforms X.

**Parameters**

- **X** (*pd.DataFrame*) – Data to fit and transform.
- **y** (*pd.Series*) – Target data.

**Returns** Transformed X.

**Return type** pd.DataFrame

**Raises** **MethodPropertyNotFoundError** – If transformer does not have a transform method or a component_obj that implements transform.

`static load`(*file_path*)

Loads component at file path.

**Parameters** **file_path** (*str*) – Location to load file.

**Returns** ComponentBase object

`needs_fitting`(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need to be fit or whose fit methods do nothing.

**Returns** True.

`property parameters`(*self*)

Returns the parameters which were used to initialize the component.

`save`(*self*, *file_path*, *pickle_protocol=cloudpickle.DEFAULT_PROTOCOL*)

Saves component at file path.

**Parameters**

- **file_path** (*str*) – Location to save file.
- **pickle_protocol** (*int*) – The pickle data stream format.

`transform`(*self*, *X*, *y=None*)

Transforms data X by imputing missing values using specified timeseries-specific strategies. 'None' values are converted to np.nan before imputation and are treated as the same.

**Parameters**

- **X** (*pd.DataFrame*) – Data to transform.
- **y** (*pd.Series, optional*) – Optionally, target data to transform.

**Returns** Transformed X and y

**Return type** pd.DataFrame

`update_parameters`(*self*, *update_dict*, *reset_fit=True*)

Updates the parameter dictionary of the component.

**Parameters**

- **update_dict** (*dict*) – A dict of parameters to update.

- **reset_fit** (*bool, optional*) – If True, will set *_is_fitted* to False.

**class** evalml.pipelines.components.transformers.**TimeSeriesRegularizer**(*time_index=None, frequency_payload=None, window_length=4, threshold=0.4, random_seed=0, **kwargs*)

Transformer that regularizes an inconsistently spaced datetime column.

If X is passed in to fit/transform, the column *time_index* will be checked for an inferrable offset frequency. If the *time_index* column is perfectly inferrable then this Transformer will do nothing and return the original X and y.

If X does not have a perfectly inferrable frequency but one can be estimated, then X and y will be reformatted based on the estimated frequency for *time_index*. In the original X and y passed: - Missing datetime values will be added and will have their corresponding columns in X and y set to None. - Duplicate datetime values will be dropped. - Extra datetime values will be dropped. - If it can be determined that a duplicate or extra value is misaligned, then it will be repositioned to take the place of a missing value.

This Transformer should be used before the *TimeSeriesImputer* in order to impute the missing values that were added to X and y (if passed).

**Parameters**

- **time_index** (*string*) – Name of the column containing the datetime information used to order the data, required. Defaults to None.
- **frequency_payload** (*tuple*) – Payload returned from Woodwork's infer_frequency function where debug is True. Defaults to None.
- **window_length** (*int*) – The size of the rolling window over which inference is conducted to determine the prevalence of uninferrable frequencies.
- **5.** (*Lower values make this component more sensitive to recognizing numerous faulty datetime values. Defaults to*) –
- **threshold** (*float*) – The minimum percentage of windows that need to have been able to infer a frequency. Lower values make this component more
- **0.8.** (*sensitive to recognizing numerous faulty datetime values. Defaults to*) –
- **random_seed** (*int*) – Seed for the random number generator. This transformer performs the same regardless of the random seed provided.
- **0.** (*Defaults to*) –

**Raises** **ValueError** – if the frequency_payload parameter has not been passed a tuple

**Attributes**

| | |
|---|---|
| **hyperparameter_ranges** | {} |
| **modifies_features** | True |
| **modifies_target** | True |
| **name** | Time Series Regularizer |
| **training_only** | True |

**Methods**

| | |
|---|---|
| *clone* | Constructs a new component with the same parameters and random state. |
| *default_parameters* | Returns the default parameters for this component. |
| *describe* | Describe a component and its parameters. |
| *fit* | Fits the TimeSeriesRegularizer. |
| *fit_transform* | Fits on X and transforms X. |
| *load* | Loads component at file path. |
| *needs_fitting* | Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances. |
| *parameters* | Returns the parameters which were used to initialize the component. |
| *save* | Saves component at file path. |
| *transform* | Regularizes a dataframe and target data to an inferrable offset frequency. |
| *update_parameters* | Updates the parameter dictionary of the component. |

**clone**(*self*)

Constructs a new component with the same parameters and random state.

**Returns** A new instance of this component with identical parameters and random state.

**default_parameters**(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

**Returns** Default parameters for this component.

**Return type** dict

**describe**(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

**Parameters**

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description as dictionary in the format {"name": name, "parameters": parameters}

**Returns** Returns dictionary if return_dict is True, else None.

**Return type** None or dict

**fit**(*self*, *X*, *y=None*)

Fits the TimeSeriesRegularizer.

**Parameters**

- **X** (*pd.DataFrame*) – The input training data of shape [n_samples, n_features].
- **y** (*pd.Series, optional*) – The target training data of length [n_samples].

**Returns** self

**Raises**

- **ValueError** – if self.time_index is None, if X and y have different lengths, if *time_index* in X does not have an offset frequency that can be estimated
- **TypeError** – if the *time_index* column is not of type Datetime
- **KeyError** – if the *time_index* column doesn't exist

**fit_transform**(*self*, *X*, *y=None*)

Fits on X and transforms X.

**Parameters**

- **X** (*pd.DataFrame*) – Data to fit and transform.
- **y** (*pd.Series*) – Target data.

**Returns** Transformed X.

**Return type** pd.DataFrame

**Raises** **MethodPropertyNotFoundError** – If transformer does not have a transform method or a component_obj that implements transform.

**static load**(*file_path*)

Loads component at file path.

**Parameters** **file_path** (*str*) – Location to load file.

**Returns** ComponentBase object

**needs_fitting**(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need to be fit or whose fit methods do nothing.

**Returns** True.

**property parameters**(*self*)

Returns the parameters which were used to initialize the component.

**save**(*self*, *file_path*, *pickle_protocol=cloudpickle.DEFAULT_PROTOCOL*)

Saves component at file path.

**Parameters**

- **file_path** (*str*) – Location to save file.
- **pickle_protocol** (*int*) – The pickle data stream format.

**transform**(*self*, *X*, *y=None*)

Regularizes a dataframe and target data to an inferrable offset frequency.

A 'clean' X and y (if y was passed in) are created based on an inferrable offset frequency and matching datetime values with the original X and y are imputed into the clean X and y. Datetime values identified as misaligned are shifted into their appropriate position.

**Parameters**

- **X** (*pd.DataFrame*) – The input training data of shape [n_samples, n_features].
- **y** (*pd.Series, optional*) – The target training data of length [n_samples].

**Returns** Data with an inferrable *time_index* offset frequency.

**Return type** (pd.DataFrame, pd.Series)

**update_parameters**(*self*, *update_dict*, *reset_fit=True*)

Updates the parameter dictionary of the component.

**Parameters**

- **update_dict** (*dict*) – A dict of parameters to update.
- **reset_fit** (*bool, optional*) – If True, will set *_is_fitted* to False.

**class** evalml.pipelines.components.transformers.**Transformer**(*parameters=None*, *component_obj=None*, *random_seed=0*, ***kwargs*)

A component that may or may not need fitting that transforms data. These components are used before an estimator.

To implement a new Transformer, define your own class which is a subclass of Transformer, including a name and a list of acceptable ranges for any parameters to be tuned during the automl search (hyperparameters). Define an *__init__* method which sets up any necessary state and objects. Make sure your *__init__* only uses standard keyword arguments and calls *super().__init__()* with a parameters dict. You may also override the *fit*, *transform*, *fit_transform* and other methods in this class if appropriate.

To see some examples, check out the definitions of any Transformer component.

**Parameters**

- **parameters** (*dict*) – Dictionary of parameters for the component. Defaults to None.
- **component_obj** (*obj*) – Third-party objects useful in component implementation. Defaults to None.
- **random_seed** (*int*) – Seed for the random number generator. Defaults to 0.

**Attributes**

| | |
|---|---|
| **modifies_features** | True |
| **modifies_target** | False |
| **training_only** | False |

**Methods**

| | |
|---|---|
| *clone* | Constructs a new component with the same parameters and random state. |
| *default_parameters* | Returns the default parameters for this component. |
| *describe* | Describe a component and its parameters. |
| *fit* | Fits component to data. |
| *fit_transform* | Fits on X and transforms X. |
| *load* | Loads component at file path. |
| *name* | Returns string name of this component. |
| *needs_fitting* | Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances. |
| *parameters* | Returns the parameters which were used to initialize the component. |
| *save* | Saves component at file path. |
| *transform* | Transforms data X. |
| *update_parameters* | Updates the parameter dictionary of the component. |

**clone**(*self*)

Constructs a new component with the same parameters and random state.

**Returns** A new instance of this component with identical parameters and random state.

**default_parameters**(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

**Returns** Default parameters for this component.

**Return type** dict

**describe**(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

**Parameters**

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description as dictionary in the format {"name": name, "parameters": parameters}

**Returns** Returns dictionary if return_dict is True, else None.

**Return type** None or dict

**fit**(*self*, *X*, *y=None*)

Fits component to data.

**Parameters**

- **X** (*pd.DataFrame*) – The input training data of shape [n_samples, n_features]
- **y** (*pd.Series, optional*) – The target training data of length [n_samples]

**Returns** self

**Raises** **MethodPropertyNotFoundError** – If component does not have a fit method or a component_obj that implements fit.

**fit_transform**(*self*, *X*, *y=None*)

Fits on X and transforms X.

**Parameters**

- **X** (*pd.DataFrame*) – Data to fit and transform.
- **y** (*pd.Series*) – Target data.

**Returns** Transformed X.

**Return type** pd.DataFrame

**Raises** **MethodPropertyNotFoundError** – If transformer does not have a transform method or a component_obj that implements transform.

**static load**(*file_path*)

Loads component at file path.

**Parameters** **file_path** (*str*) – Location to load file.

**Returns** ComponentBase object

**property name**(*cls*)

Returns string name of this component.

**needs_fitting**(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need to be fit or whose fit methods do nothing.

**Returns** True.

**property parameters**(*self*)

Returns the parameters which were used to initialize the component.

**save**(*self*, *file_path*, *pickle_protocol=cloudpickle.DEFAULT_PROTOCOL*)

Saves component at file path.

**Parameters**

- **file_path** (*str*) – Location to save file.
- **pickle_protocol** (*int*) – The pickle data stream format.

**abstract transform**(*self*, *X*, *y=None*)

Transforms data X.

**Parameters**

- **X** (*pd.DataFrame*) – Data to transform.
- **y** (*pd.Series, optional*) – Target data.

**Returns** Transformed X

**Return type** pd.DataFrame

**Raises** **MethodPropertyNotFoundError** – If transformer does not have a transform method or a component_obj that implements transform.

**update_parameters**(*self*, *update_dict*, *reset_fit=True*)

Updates the parameter dictionary of the component.

**Parameters**

- **update_dict** (*dict*) – A dict of parameters to update.
- **reset_fit** (*bool, optional*) – If True, will set *_is_fitted* to False.

**class** evalml.pipelines.components.transformers.**Undersampler**(*sampling_ratio=0.25*, *sampling_ratio_dict=None*, *min_samples=100*, *min_percentage=0.1*, *random_seed=0*, ***kwargs*)

Initializes an undersampling transformer to downsample the majority classes in the dataset.

This component is only run during training and not during predict.

**Parameters**

- **sampling_ratio** (*float*) – The smallest minority:majority ratio that is accepted as 'balanced'. For instance, a 1:4 ratio would be represented as 0.25, while a 1:1 ratio is 1.0. Must be between 0 and 1, inclusive. Defaults to 0.25.
- **sampling_ratio_dict** (*dict*) – A dictionary specifying the desired balanced ratio for each target value. For instance, in a binary case where class 1 is the minority, we could specify: *sampling_ratio_dict={0: 0.5, 1: 1}*, which means we would undersample class 0 to have twice the number of samples as class 1 (minority:majority ratio = 0.5), and don't sample class 1. Overrides sampling_ratio if provided. Defaults to None.
- **min_samples** (*int*) – The minimum number of samples that we must have for any class, pre or post sampling. If a class must be downsampled, it will not be downsampled past this value. To determine severe imbalance, the minority class must occur less often than this and must have a class ratio below min_percentage. Must be greater than 0. Defaults to 100.
- **min_percentage** (*float*) – The minimum percentage of the minimum class to total dataset that we tolerate, as long as it is above min_samples. If min_percentage and min_samples are not met, treat this as severely imbalanced, and we will not resample the data. Must be between 0 and 0.5, inclusive. Defaults to 0.1.
- **random_seed** (*int*) – The seed to use for random sampling. Defaults to 0.

**Raises**

- **ValueError** – If sampling_ratio is not in the range (0, 1].
- **ValueError** – If min_sample is not greater than 0.
- **ValueError** – If min_percentage is not between 0 and 0.5, inclusive.

**Attributes**

| | |
|---|---|
| **hyper-parameter_ranges** | {} |
| **modifies_features** | True |
| **modifies_target** | True |
| **name** | Undersampler |
| **training_only** | True |

**Methods**

| | |
|---|---|
| *clone* | Constructs a new component with the same parameters and random state. |
| *default_parameters* | Returns the default parameters for this component. |
| *describe* | Describe a component and its parameters. |
| *fit* | Fits the sampler to the data. |
| *fit_resample* | Resampling technique for this sampler. |
| *fit_transform* | Fit and transform data using the sampler component. |
| *load* | Loads component at file path. |
| *needs_fitting* | Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances. |
| *parameters* | Returns the parameters which were used to initialize the component. |
| *save* | Saves component at file path. |
| *transform* | Transforms the input data by sampling the data. |
| *update_parameters* | Updates the parameter dictionary of the component. |

**clone**(*self*)

Constructs a new component with the same parameters and random state.

**Returns** A new instance of this component with identical parameters and random state.

**default_parameters**(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

**Returns** Default parameters for this component.

**Return type** dict

**describe**(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

**Parameters**

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description as dictionary in the format {"name": name, "parameters": parameters}

**Returns** Returns dictionary if return_dict is True, else None.

**Return type** None or dict

**fit**(*self*, *X*, *y*)

Fits the sampler to the data.

**Parameters**

- **X** (*pd.DataFrame*) – Input features.
- **y** (*pd.Series*) – Target.

**Returns** self

**Raises** **ValueError** – If y is None.

**fit_resample**(*self*, *X*, *y*)

Resampling technique for this sampler.

**Parameters**

- **X** (*pd.DataFrame*) – Training data to fit and resample.
- **y** (*pd.Series*) – Training data targets to fit and resample.

**Returns** Indices to keep for training data.

**Return type** list

**fit_transform**(*self*, *X*, *y*)

Fit and transform data using the sampler component.

**Parameters**

- **X** (*pd.DataFrame*) – The input training data of shape [n_samples, n_features].
- **y** (*pd.Series, optional*) – The target training data of length [n_samples].

**Returns** Transformed data.

**Return type** (pd.DataFrame, pd.Series)

**static load**(*file_path*)

Loads component at file path.

**Parameters** **file_path** (*str*) – Location to load file.

**Returns** ComponentBase object

**needs_fitting**(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need to be fit or whose fit methods do nothing.

**Returns** True.

**property parameters**(*self*)

Returns the parameters which were used to initialize the component.

**save**(*self*, *file_path*, *pickle_protocol=cloudpickle.DEFAULT_PROTOCOL*)

Saves component at file path.

**Parameters**

- **file_path** (*str*) – Location to save file.
- **pickle_protocol** (*int*) – The pickle data stream format.

**transform**(*self*, *X*, *y=None*)

Transforms the input data by sampling the data.

**Parameters**

- **X** (*pd.DataFrame*) – Training features.
- **y** (*pd.Series*) – Target.

**Returns** Transformed features and target.

**Return type** pd.DataFrame, pd.Series

**update_parameters**(*self*, *update_dict*, *reset_fit=True*)

Updates the parameter dictionary of the component.

**Parameters**

- **update_dict** (*dict*) – A dict of parameters to update.
- **reset_fit** (*bool, optional*) – If True, will set *_is_fitted* to False.

**class** evalml.pipelines.components.transformers.**URLFeaturizer**(*random_seed=0*, ***kwargs*)

Transformer that can automatically extract features from URL.

**Parameters** **random_seed** (*int*) – Seed for the random number generator. Defaults to 0.

**Attributes**

| | |
|---|---|
| **hyper-parameter_ranges** | {} |
| **modifies_features** | True |
| **modifies_target** | False |
| **name** | URL Featurizer |
| **training_only** | False |

**Methods**

| | |
|---|---|
| *clone* | Constructs a new component with the same parameters and random state. |
| *default_parameters* | Returns the default parameters for this component. |
| *describe* | Describe a component and its parameters. |
| *fit* | Fits component to data. |
| *fit_transform* | Fits on X and transforms X. |
| *load* | Loads component at file path. |
| *needs_fitting* | Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances. |
| *parameters* | Returns the parameters which were used to initialize the component. |
| *save* | Saves component at file path. |
| *transform* | Transforms data X. |
| *update_parameters* | Updates the parameter dictionary of the component. |

**clone**(*self*)

Constructs a new component with the same parameters and random state.

**Returns** A new instance of this component with identical parameters and random state.

**default_parameters**(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

**Returns** Default parameters for this component.

**Return type** dict

**describe**(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

**Parameters**

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description as dictionary in the format {"name": name, "parameters": parameters}

**Returns** Returns dictionary if return_dict is True, else None.

**Return type** None or dict

**fit**(*self*, *X*, *y=None*)

Fits component to data.

**Parameters**

- **X** (*pd.DataFrame*) – The input training data of shape [n_samples, n_features]
- **y** (*pd.Series, optional*) – The target training data of length [n_samples]

**Returns** self

**Raises** **MethodPropertyNotFoundError** – If component does not have a fit method or a component_obj that implements fit.

**fit_transform**(*self*, *X*, *y=None*)

Fits on X and transforms X.

**Parameters**

- **X** (*pd.DataFrame*) – Data to fit and transform.
- **y** (*pd.Series*) – Target data.

**Returns** Transformed X.

**Return type** pd.DataFrame

**Raises** **MethodPropertyNotFoundError** – If transformer does not have a transform method or a component_obj that implements transform.

**static load**(*file_path*)

Loads component at file path.

**Parameters** **file_path** (*str*) – Location to load file.

**Returns** ComponentBase object

**needs_fitting**(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need to be fit or whose fit methods do nothing.

**Returns** True.

**property parameters**(*self*)

Returns the parameters which were used to initialize the component.

**save**(*self*, *file_path*, *pickle_protocol=cloudpickle.DEFAULT_PROTOCOL*)

Saves component at file path.

**Parameters**

- **file_path** (*str*) – Location to save file.
- **pickle_protocol** (*int*) – The pickle data stream format.

**transform**(*self*, *X*, *y=None*)

Transforms data X.

**Parameters**

- **X** (*pd.DataFrame*) – Data to transform.
- **y** (*pd.Series, optional*) – Target data.

**Returns** Transformed X

**Return type** pd.DataFrame

**Raises** **MethodPropertyNotFoundError** – If transformer does not have a transform method or a component_obj that implements transform.

**update_parameters**(*self*, *update_dict*, *reset_fit=True*)

Updates the parameter dictionary of the component.

**Parameters**

- **update_dict** (*dict*) – A dict of parameters to update.
- **reset_fit** (*bool, optional*) – If True, will set *_is_fitted* to False.

## Submodules

### component_base

Base class for all components.

## Module Contents

## Classes Summary

| | |
|---|---|
| *ComponentBase* | Base class for all components. |

## Contents

**class** evalml.pipelines.components.component_base.**ComponentBase**(*parameters=None*, *component_obj=None*, *random_seed=0*, ***kwargs*)

Base class for all components.

**Parameters**

- **parameters** (*dict*) – Dictionary of parameters for the component. Defaults to None.
- **component_obj** (*obj*) – Third-party objects useful in component implementation. Defaults to None.
- **random_seed** (*int*) – Seed for the random number generator. Defaults to 0.

**Methods**

| | |
|---|---|
| *clone* | Constructs a new component with the same parameters and random state. |
| *default_parameters* | Returns the default parameters for this component. |
| *describe* | Describe a component and its parameters. |
| *fit* | Fits component to data. |
| *load* | Loads component at file path. |
| *modifies_features* | Returns whether this component modifies (subsets or transforms) the features variable during transform. |
| *modifies_target* | Returns whether this component modifies (subsets or transforms) the target variable during transform. |
| *name* | Returns string name of this component. |
| *needs_fitting* | Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances. |
| *parameters* | Returns the parameters which were used to initialize the component. |
| *save* | Saves component at file path. |
| *training_only* | Returns whether or not this component should be evaluated during training-time only, or during both training and prediction time. |
| *update_parameters* | Updates the parameter dictionary of the component. |

**clone**(*self*)

Constructs a new component with the same parameters and random state.

**Returns** A new instance of this component with identical parameters and random state.

**default_parameters**(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

**Returns** Default parameters for this component.

**Return type** dict

**describe**(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

**Parameters**

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description as dictionary in the format {"name": name, "parameters": parameters}

**Returns** Returns dictionary if return_dict is True, else None.

**Return type** None or dict

**fit**(*self*, *X*, *y=None*)

Fits component to data.

**Parameters**

- **X** (*pd.DataFrame*) – The input training data of shape [n_samples, n_features]
- **y** (*pd.Series, optional*) – The target training data of length [n_samples]

**Returns** self

**Raises** **MethodPropertyNotFoundError** – If component does not have a fit method or a component_obj that implements fit.

**static load**(*file_path*)

Loads component at file path.

**Parameters** **file_path** (*str*) – Location to load file.

**Returns** ComponentBase object

**property modifies_features**(*cls*)

Returns whether this component modifies (subsets or transforms) the features variable during transform.

For Estimator objects, this attribute determines if the return value from *predict* or *predict_proba* should be used as features or targets.

**property modifies_target**(*cls*)

Returns whether this component modifies (subsets or transforms) the target variable during transform.

For Estimator objects, this attribute determines if the return value from *predict* or *predict_proba* should be used as features or targets.

**property name**(*cls*)

Returns string name of this component.

**needs_fitting**(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need to be fit or whose fit methods do nothing.

**Returns** True.

**property parameters**(*self*)

Returns the parameters which were used to initialize the component.

**save**(*self*, *file_path*, *pickle_protocol=cloudpickle.DEFAULT_PROTOCOL*)

Saves component at file path.

**Parameters**

- **file_path** (*str*) – Location to save file.
- **pickle_protocol** (*int*) – The pickle data stream format.

**property training_only**(*cls*)

Returns whether or not this component should be evaluated during training-time only, or during both training and prediction time.

**update_parameters**(*self*, *update_dict*, *reset_fit=True*)

Updates the parameter dictionary of the component.

**Parameters**

- **update_dict** (*dict*) – A dict of parameters to update.
- **reset_fit** (*bool, optional*) – If True, will set *_is_fitted* to False.

## component_base_meta

Metaclass that overrides creating a new component by wrapping methods with validators and setters.

### Module Contents

### Classes Summary

| | |
|---|---|
| *ComponentBaseMeta* | Metaclass that overrides creating a new component by wrapping methods with validators and setters. |

### Contents

**class** evalml.pipelines.components.component_base_meta.**ComponentBaseMeta**

Metaclass that overrides creating a new component by wrapping methods with validators and setters.

**Attributes**

| | |
|---|---|
| **FIT_METHODS** | ['fit', 'fit_transform'] |
| **METHODS_TO_CHECK** | ['predict', 'predict_proba', 'transform', 'inverse_transform', 'get_trend_dataframe'] |
| **PROPERTIES_TO_CHECK** | ['feature_importance'] |

**Methods**

| | |
|---|---|
| `check_for_fit` | *check_for_fit* wraps a method that validates if *self._is_fitted* is *True*. |
| `register` | Register a virtual subclass of an ABC. |
| `set_fit` | Wrapper for the fit method. |

**classmethod check_for_fit**(*cls*, *method*)

*check_for_fit* wraps a method that validates if *self._is_fitted* is *True*.

It raises an exception if *False* and calls and returns the wrapped method if *True*.

**Parameters** **method** (*callable*) – Method to wrap.

**Returns** The wrapped method.

**Raises** **ComponentNotYetFittedError** – If component is not yet fitted.

**register**(*cls*, *subclass*)

Register a virtual subclass of an ABC.

Returns the subclass, to allow usage as a class decorator.

**classmethod set_fit**(*cls*, *method*)

Wrapper for the fit method.

### utils

Utility methods for EvalML components.

### Module Contents

### Classes Summary

| | |
|---|---|
| `WrappedSKClassifier` | Scikit-learn classifier wrapper class. |
| `WrappedSKRegressor` | Scikit-learn regressor wrapper class. |

### Functions

| | |
|---|---|
| *all_components* | Get all available components. |
| *allowed_model_families* | List the model types allowed for a particular problem type. |
| *estimator_unable_to_handle_nans* | If True, provided estimator class is unable to handle NaN values as an input. |
| *generate_component_code* | Creates and returns a string that contains the Python imports and code required for running the EvalML component. |
| *get_estimators* | Returns the estimators allowed for a particular problem type. |
| *get_prediction_intevals_for_tree_regressors* | Find the prediction intervals for tree-based regressors. |
| *handle_component_class* | Standardizes input from a string name to a ComponentBase subclass if necessary. |
| *handle_float_categories_for_catboost* | Updates input data to be compatible with CatBoost estimators. |
| *make_balancing_dictionary* | Makes dictionary for oversampler components. Find ratio of each class to the majority. If the ratio is smaller than the sampling_ratio, we want to oversample, otherwise, we don't want to sample at all, and we leave the data as is. |
| *scikit_learn_wrapped_estimator* | Wraps an EvalML object as a scikit-learn estimator. |

### Contents

evalml.pipelines.components.utils.**all_components**()

Get all available components.

evalml.pipelines.components.utils.**allowed_model_families**(*problem_type*)

List the model types allowed for a particular problem type.

**Parameters** **problem_type** (*ProblemTypes or str*) – ProblemTypes enum or string.

**Returns** A list of model families.

**Return type** list[ModelFamily]

evalml.pipelines.components.utils.**estimator_unable_to_handle_nans**(*estimator_class*)

If True, provided estimator class is unable to handle NaN values as an input.

**Parameters** **estimator_class** (*Estimator*) – Estimator class

**Raises** **ValueError** – If estimator is not a valid estimator class.

**Returns** True if estimator class is unable to process NaN values, False otherwise.

**Return type** bool

evalml.pipelines.components.utils.**generate_component_code**(*element*)

Creates and returns a string that contains the Python imports and code required for running the EvalML component.

**Parameters** **element** (*component instance*) – The instance of the component to generate string Python code for.

**Returns** String representation of Python code that can be run separately in order to recreate the component instance. Does not include code for custom component implementation.

**Raises** **ValueError** – If the input element is not a component instance.

### Examples

```
>>> from evalml.pipelines.components.estimators.regressors.decision_tree_regressor␣
↪import DecisionTreeRegressor
>>> assert generate_component_code(DecisionTreeRegressor()) == "from evalml.
↪pipelines.components.estimators.regressors.decision_tree_regressor import␣
↪DecisionTreeRegressor\n\ndecisionTreeRegressor = DecisionTreeRegressor(**{
↪'criterion': 'squared_error', 'max_features': 'auto', 'max_depth': 6, 'min_
↪samples_split': 2, 'min_weight_fraction_leaf': 0.0})"
...
>>> from evalml.pipelines.components.transformers.imputers.simple_imputer import␣
↪SimpleImputer
>>> assert generate_component_code(SimpleImputer()) == "from evalml.pipelines.
↪components.transformers.imputers.simple_imputer import SimpleImputer\n\
↪nsimpleImputer = SimpleImputer(**{'impute_strategy': 'most_frequent', 'fill_value
↪': None})"
```

`evalml.pipelines.components.utils.`**`get_estimators`**(*problem_type*, *model_families=None*)

Returns the estimators allowed for a particular problem type.

Can also optionally filter by a list of model types.

**Parameters**

- **problem_type** (*ProblemTypes or str*) – Problem type to filter for.
- **model_families** (*list[ModelFamily] or list[str]*) – Model families to filter for.

**Returns** A list of estimator subclasses.

**Return type** list[class]

**Raises**

- **TypeError** – If the model_families parameter is not a list.
- **RuntimeError** – If a model family is not valid for the problem type.

`evalml.pipelines.components.utils.`**`get_prediction_intevals_for_tree_regressors`**(*X: pandas.DataFrame*, *predictions: pandas.Series*, *coverage: List[float]*, *estimators: List[*evalml.pipelines.components → Dict[str, pandas.Series]

Find the prediction intervals for tree-based regressors.

**Parameters**

- **X** (*pd.DataFrame*) – Data of shape [n_samples, n_features].

- **predictions** (*pd.Series*) – Predictions from the regressor.
- **coverage** (*list[float]*) – A list of floats between the values 0 and 1 that the upper and lower bounds of the prediction interval should be calculated for.
- **estimators** (*list*) – Collection of fitted sub-estimators.

**Returns** Prediction intervals, keys are in the format {coverage}_lower or {coverage}_upper.

**Return type** dict

`evalml.pipelines.components.utils.`**`handle_component_class`**(*component_class*)

Standardizes input from a string name to a ComponentBase subclass if necessary.

If a str is provided, will attempt to look up a ComponentBase class by that name and return a new instance. Otherwise if a ComponentBase subclass or Component instance is provided, will return that without modification.

**Parameters** **component_class** (*str, ComponentBase*) – Input to be standardized.

**Returns** ComponentBase

**Raises**

- **ValueError** – If input is not a valid component class.
- **MissingComponentError** – If the component cannot be found.

### Examples

```
>>> from evalml.pipelines.components.estimators.regressors.decision_tree_regressor␣
↪import DecisionTreeRegressor
>>> handle_component_class(DecisionTreeRegressor)
<class 'evalml.pipelines.components.estimators.regressors.decision_tree_regressor.
↪DecisionTreeRegressor'>
>>> handle_component_class("Random Forest Regressor")
<class 'evalml.pipelines.components.estimators.regressors.rf_regressor.
↪RandomForestRegressor'>
```

`evalml.pipelines.components.utils.`**`handle_float_categories_for_catboost`**(*X*)

Updates input data to be compatible with CatBoost estimators.

CatBoost cannot handle data in X that is the Categorical Woodwork logical type with floating point categories. This utility determines if the floating point categories can be converted to integers without truncating any data, and if they can be, converts them to int64 categories. Will not attempt to use values that are truly floating points.

**Parameters** **X** (*pd.DataFrame*) – Input data to CatBoost that has Woodwork initialized

**Returns**

**Input data with exact same Woodwork typing info as the original but with any float categories** converted to be int64 when possible.

**Return type** DataFrame

**Raises** **ValueError** – if the numeric categories are actual floats that cannot be converted to integers without truncating data

`evalml.pipelines.components.utils.`**`make_balancing_dictionary`**(*y, sampling_ratio*)

Makes dictionary for oversampler components. Find ratio of each class to the majority. If the ratio is smaller than the sampling_ratio, we want to oversample, otherwise, we don't want to sample at all, and we leave the data as is.

**Parameters**

- **y** (*pd.Series*) – Target data.
- **sampling_ratio** (*float*) – The balanced ratio we want the samples to meet.

**Returns** Dictionary where keys are the classes, and the corresponding values are the counts of samples for each class that will satisfy sampling_ratio.

**Return type** dict

**Raises** **ValueError** – If sampling ratio is not in the range (0, 1] or the target is empty.

### Examples

```
>>> import pandas as pd
>>> y = pd.Series([1] * 4 + [2] * 8 + [3])
>>> assert make_balancing_dictionary(y, 0.5) == {2: 8, 1: 4, 3: 4}
>>> assert make_balancing_dictionary(y, 0.9) == {2: 8, 1: 7, 3: 7}
>>> assert make_balancing_dictionary(y, 0.1) == {2: 8, 1: 4, 3: 1}
```

evalml.pipelines.components.utils.**scikit_learn_wrapped_estimator**(*evalml_obj*)

Wraps an EvalML object as a scikit-learn estimator.

**class** evalml.pipelines.components.utils.**WrappedSKClassifier**(*pipeline*)

Scikit-learn classifier wrapper class.

**Methods**

| | |
|---|---|
| *fit* | Fits component to data. |
| *get_params* | Get parameters for this estimator. |
| *predict* | Make predictions using selected features. |
| *predict_proba* | Make probability estimates for labels. |
| *score* | Return the mean accuracy on the given test data and labels. |
| *set_params* | Set the parameters of this estimator. |

**fit**(*self*, *X*, *y*)

Fits component to data.

**Parameters**

- **X** (*pd.DataFrame or np.ndarray*) – The input training data of shape [n_samples, n_features].
- **y** (*pd.Series, optional*) – The target training data of length [n_samples].

**Returns** self

**get_params**(*self*, *deep=True*)

Get parameters for this estimator.

**Parameters** **deep** (*bool, default=True*) – If True, will return the parameters for this estimator and contained subobjects that are estimators.

**Returns** **params** – Parameter names mapped to their values.

**Return type** dict

**predict**(*self*, *X*)

Make predictions using selected features.

**Parameters** **X** (*pd.DataFrame*) – Features

**Returns** Predicted values.

**Return type** np.ndarray

**predict_proba**(*self*, *X*)

Make probability estimates for labels.

**Parameters** **X** (*pd.DataFrame*) – Features.

**Returns** Probability estimates.

**Return type** np.ndarray

**score**(*self*, *X*, *y*, *sample_weight=None*)

Return the mean accuracy on the given test data and labels.

In multi-label classification, this is the subset accuracy which is a harsh metric since you require for each sample that each label set be correctly predicted.

**Parameters**

- **X** (*array-like of shape (n_samples, n_features)*) – Test samples.
- **y** (*array-like of shape (n_samples,) or (n_samples, n_outputs)*) – True labels for *X*.
- **sample_weight** (*array-like of shape (n_samples,), default=None*) – Sample weights.

**Returns** **score** – Mean accuracy of `self.predict(X)` w.r.t. *y*.

**Return type** float

**set_params**(*self*, ***params*)

Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as `Pipeline`). The latter have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

**Parameters** ****params** (*dict*) – Estimator parameters.

**Returns** **self** – Estimator instance.

**Return type** estimator instance

**class** evalml.pipelines.components.utils.**WrappedSKRegressor**(*pipeline*)

Scikit-learn regressor wrapper class.

**Methods**

| | |
|---|---|
| *fit* | Fits component to data. |
| *get_params* | Get parameters for this estimator. |
| *predict* | Make predictions using selected features. |
| *score* | Return the coefficient of determination of the prediction. |
| *set_params* | Set the parameters of this estimator. |

**fit**(*self*, *X*, *y*)

Fits component to data.

**Parameters**

- **X** (*pd.DataFrame or np.ndarray*) – the input training data of shape [n_samples, n_features]
- **y** (*pd.Series, optional*) – the target training data of length [n_samples]

**Returns** self

**get_params**(*self*, *deep=True*)

Get parameters for this estimator.

**Parameters** **deep** (*bool, default=True*) – If True, will return the parameters for this estimator and contained subobjects that are estimators.

**Returns** **params** – Parameter names mapped to their values.

**Return type** dict

**predict**(*self*, *X*)

Make predictions using selected features.

**Parameters** **X** (*pd.DataFrame*) – Features.

**Returns** Predicted values.

**Return type** np.ndarray

**score**(*self*, *X*, *y*, *sample_weight=None*)

Return the coefficient of determination of the prediction.

The coefficient of determination $R^2$ is defined as $(1 - \frac{u}{v})$, where $u$ is the residual sum of squares `((y_true - y_pred)** 2).sum()` and $v$ is the total sum of squares `((y_true - y_true.mean()) ** 2).sum()`. The best possible score is 1.0 and it can be negative (because the model can be arbitrarily worse). A constant model that always predicts the expected value of *y*, disregarding the input features, would get a $R^2$ score of 0.0.

**Parameters**

- **X** (*array-like of shape (n_samples, n_features)*) – Test samples. For some estimators this may be a precomputed kernel matrix or a list of generic objects instead with shape `(n_samples, n_samples_fitted)`, where `n_samples_fitted` is the number of samples used in the fitting for the estimator.
- **y** (*array-like of shape (n_samples,) or (n_samples, n_outputs)*) – True values for *X*.
- **sample_weight** (*array-like of shape (n_samples,), default=None*) – Sample weights.

**Returns** **score** – $R^2$ of `self.predict(X)` w.r.t. $y$.

**Return type** float

### Notes

The $R^2$ score used when calling `score` on a regressor uses `multioutput='uniform_average'` from version 0.23 to keep consistent with default value of `r2_score()`. This influences the `score` method of all the multioutput regressors (except for `MultiOutputRegressor`).

**set_params**(*self*, ***params*)

Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as `Pipeline`). The latter have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

**Parameters** ****params** (*dict*) – Estimator parameters.

**Returns** **self** – Estimator instance.

**Return type** estimator instance

## Package Contents

## Classes Summary

| | |
|---|---|
| *ARIMARegressor* | Autoregressive Integrated Moving Average Model. The three parameters (p, d, q) are the AR order, the degree of differencing, and the MA order. More information here: https://www.statsmodels.org/devel/generated/statsmodels.tsa.arima.model.ARIMA.html. |
| *BaselineClassifier* | Classifier that predicts using the specified strategy. |
| *BaselineRegressor* | Baseline regressor that uses a simple strategy to make predictions. This is useful as a simple baseline regressor to compare with other regressors. |
| *CatBoostClassifier* | CatBoost Classifier, a classifier that uses gradient-boosting on decision trees. CatBoost is an open-source library and natively supports categorical features. |
| *CatBoostRegressor* | CatBoost Regressor, a regressor that uses gradient-boosting on decision trees. CatBoost is an open-source library and natively supports categorical features. |
| *ComponentBase* | Base class for all components. |
| *ComponentBaseMeta* | Metaclass that overrides creating a new component by wrapping methods with validators and setters. |
| *DateTimeFeaturizer* | Transformer that can automatically extract features from datetime columns. |
| *DecisionTreeClassifier* | Decision Tree Classifier. |
| *DecisionTreeRegressor* | Decision Tree Regressor. |
| *DFSTransformer* | Featuretools DFS component that generates features for the input features. |
| *DropColumns* | Drops specified columns in input data. |
| *DropNaNRowsTransformer* | Transformer to drop rows with NaN values. |
| *DropNullColumns* | Transformer to drop features whose percentage of NaN values exceeds a specified threshold. |
| *DropRowsTransformer* | Transformer to drop rows specified by row indices. |

continues on next page

Table 6 – continued from previous page

| | |
|---|---|
| *ElasticNetClassifier* | Elastic Net Classifier. Uses Logistic Regression with elasticnet penalty as the base estimator. |
| *ElasticNetRegressor* | Elastic Net Regressor. |
| *EmailFeaturizer* | Transformer that can automatically extract features from emails. |
| *Estimator* | A component that fits and predicts given data. |
| *ExponentialSmoothingRegressor* | Holt-Winters Exponential Smoothing Forecaster. |
| *ExtraTreesClassifier* | Extra Trees Classifier. |
| *ExtraTreesRegressor* | Extra Trees Regressor. |
| *FeatureSelector* | Selects top features based on importance weights. |
| *Imputer* | Imputes missing data according to a specified imputation strategy. |
| *KNeighborsClassifier* | K-Nearest Neighbors Classifier. |
| *LabelEncoder* | A transformer that encodes target labels using values between 0 and num_classes - 1. |
| *LightGBMClassifier* | LightGBM Classifier. |
| *LightGBMRegressor* | LightGBM Regressor. |
| *LinearDiscriminantAnalysis* | Reduces the number of features by using Linear Discriminant Analysis. |
| *LinearRegressor* | Linear Regressor. |
| *LogisticRegressionClassifier* | Logistic Regression Classifier. |
| *LogTransformer* | Applies a log transformation to the target data. |
| *LSA* | Transformer to calculate the Latent Semantic Analysis Values of text input. |
| *NaturalLanguageFeaturizer* | Transformer that can automatically featurize text columns using featuretools' nlp_primitives. |
| *OneHotEncoder* | A transformer that encodes categorical features in a one-hot numeric array. |
| *OrdinalEncoder* | A transformer that encodes ordinal features as an array of ordinal integers representing the relative order of categories. |
| *Oversampler* | SMOTE Oversampler component. Will automatically select whether to use SMOTE, SMOTEN, or SMOTENC based on inputs to the component. |
| *PCA* | Reduces the number of features by using Principal Component Analysis (PCA). |
| *PerColumnImputer* | Imputes missing data according to a specified imputation strategy per column. |
| *PolynomialDecomposer* | Removes trends and seasonality from time series by fitting a polynomial and moving average to the data. |
| *ProphetRegressor* | Prophet is a procedure for forecasting time series data based on an additive model where non-linear trends are fit with yearly, weekly, and daily seasonality, plus holiday effects. It works best with time series that have strong seasonal effects and several seasons of historical data. Prophet is robust to missing data and shifts in the trend, and typically handles outliers well. |
| *RandomForestClassifier* | Random Forest Classifier. |
| *RandomForestRegressor* | Random Forest Regressor. |
| *ReplaceNullableTypes* | Transformer to replace features with the new nullable dtypes with a dtype that is compatible in EvalML. |

continues on next page

Table 6 – continued from previous page

| | |
|---|---|
| RFClassifierRFESelector | Selects relevant features using recursive feature elimination with a Random Forest Classifier. |
| RFClassifierSelectFromModel | Selects top features based on importance weights using a Random Forest classifier. |
| RFRegressorRFESelector | Selects relevant features using recursive feature elimination with a Random Forest Regressor. |
| RFRegressorSelectFromModel | Selects top features based on importance weights using a Random Forest regressor. |
| SelectByType | Selects columns by specified Woodwork logical type or semantic tag in input data. |
| SelectColumns | Selects specified columns in input data. |
| SimpleImputer | Imputes missing data according to a specified imputation strategy. Natural language columns are ignored. |
| StackedEnsembleBase | Stacked Ensemble Base Class. |
| StackedEnsembleClassifier | Stacked Ensemble Classifier. |
| StackedEnsembleRegressor | Stacked Ensemble Regressor. |
| StandardScaler | A transformer that standardizes input features by removing the mean and scaling to unit variance. |
| STLDecomposer | Removes trends and seasonality from time series using the STL algorithm. |
| SVMClassifier | Support Vector Machine Classifier. |
| SVMRegressor | Support Vector Machine Regressor. |
| TargetEncoder | A transformer that encodes categorical features into target encodings. |
| TargetImputer | Imputes missing target data according to a specified imputation strategy. |
| TimeSeriesBaselineEstimator | Time series estimator that predicts using the naive forecasting approach. |
| TimeSeriesFeaturizer | Transformer that delays input features and target variable for time series problems. |
| TimeSeriesImputer | Imputes missing data according to a specified timeseries-specific imputation strategy. |
| TimeSeriesRegularizer | Transformer that regularizes an inconsistently spaced datetime column. |
| Transformer | A component that may or may not need fitting that transforms data. These components are used before an estimator. |
| Undersampler | Initializes an undersampling transformer to downsample the majority classes in the dataset. |
| URLFeaturizer | Transformer that can automatically extract features from URL. |
| VowpalWabbitBinaryClassifier | Vowpal Wabbit Binary Classifier. |
| VowpalWabbitMulticlassClassifier | Vowpal Wabbit Multiclass Classifier. |
| VowpalWabbitRegressor | Vowpal Wabbit Regressor. |
| XGBoostClassifier | XGBoost Classifier. |
| XGBoostRegressor | XGBoost Regressor. |

### Contents

**class** evalml.pipelines.components.**ARIMARegressor**(*time_index: Optional[Hashable] = None, trend: Optional[str] = None, start_p: int = 2, d: int = 0, start_q: int = 2, max_p: int = 5, max_d: int = 2, max_q: int = 5, seasonal: bool = True, sp: int = 1, n_jobs: int = - 1, random_seed: Union[int, float] = 0, maxiter: int = 10, use_covariates: bool = True, **kwargs*)

Autoregressive Integrated Moving Average Model. The three parameters (p, d, q) are the AR order, the degree of differencing, and the MA order. More information here: https://www.statsmodels.org/devel/generated/statsmodels.tsa.arima.model.ARIMA.html.

Currently ARIMARegressor isn't supported via conda install. It's recommended that it be installed via PyPI.

**Parameters**

- **time_index** (*str*) – Specifies the name of the column in X that provides the datetime objects. Defaults to None.
- **trend** (*str*) – Controls the deterministic trend. Options are ['n', 'c', 't', 'ct'] where 'c' is a constant term, 't' indicates a linear trend, and 'ct' is both. Can also be an iterable when defining a polynomial, such as [1, 1, 0, 1].
- **start_p** (*int*) – Minimum Autoregressive order. Defaults to 2.
- **d** (*int*) – Minimum Differencing degree. Defaults to 0.
- **start_q** (*int*) – Minimum Moving Average order. Defaults to 2.
- **max_p** (*int*) – Maximum Autoregressive order. Defaults to 5.
- **max_d** (*int*) – Maximum Differencing degree. Defaults to 2.
- **max_q** (*int*) – Maximum Moving Average order. Defaults to 5.
- **seasonal** (*boolean*) – Whether to fit a seasonal model to ARIMA. Defaults to True.
- **sp** (*int or str*) – Period for seasonal differencing, specifically the number of periods in each season. If "detect", this model will automatically detect this parameter (given the time series is a standard frequency) and will fall back to 1 (no seasonality) if it cannot be detected. Defaults to 1.
- **n_jobs** (*int or None*) – Non-negative integer describing level of parallelism used for pipelines. Defaults to -1.
- **random_seed** (*int*) – Seed for the random number generator. Defaults to 0.

**Attributes**

| | |
|---|---|
| **hyper-parameter_ranges** | { "start_p": Integer(1, 3), "d": Integer(0, 2), "start_q": Integer(1, 3), "max_p": Integer(3, 10), "max_d": Integer(2, 5), "max_q": Integer(3, 10), "seasonal": [True, False],} |
| **max_cols** | 7 |
| **max_rows** | 1000 |
| **model_family** | ModelFamily.ARIMA |
| **modifies_features** | True |
| **modifies_target** | False |
| **name** | ARIMA Regressor |
| **supported_problem_types** | [ProblemTypes.TIME_SERIES_REGRESSION] |
| **training_only** | False |

**Methods**

| | |
|---|---|
| *clone* | Constructs a new component with the same parameters and random state. |
| *default_parameters* | Returns the default parameters for this component. |
| *describe* | Describe a component and its parameters. |
| *feature_importance* | Returns array of 0's with a length of 1 as feature_importance is not defined for ARIMA regressor. |
| *fit* | Fits ARIMA regressor to data. |
| *get_prediction_intervals* | Find the prediction intervals using the fitted ARIMARegressor. |
| *load* | Loads component at file path. |
| *needs_fitting* | Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances. |
| *parameters* | Returns the parameters which were used to initialize the component. |
| *predict* | Make predictions using fitted ARIMA regressor. |
| *predict_proba* | Make probability estimates for labels. |
| *save* | Saves component at file path. |
| *update_parameters* | Updates the parameter dictionary of the component. |

**clone**(*self*)

Constructs a new component with the same parameters and random state.

**Returns** A new instance of this component with identical parameters and random state.

**default_parameters**(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

**Returns** Default parameters for this component.

**Return type** dict

**describe**(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

**Parameters**

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description as dictionary in the format {"name": name, "parameters": parameters}

**Returns** Returns dictionary if return_dict is True, else None.

**Return type** None or dict

**property feature_importance**(*self*) → numpy.ndarray

Returns array of 0's with a length of 1 as feature_importance is not defined for ARIMA regressor.

**fit**(*self, X: pandas.DataFrame, y: Optional[pandas.Series] = None*)

Fits ARIMA regressor to data.

**Parameters**

- **X** (*pd.DataFrame*) – The input training data of shape [n_samples, n_features].
- **y** (*pd.Series*) – The target training data of length [n_samples].

**Returns** self

**Raises** **ValueError** – If y was not passed in.

**get_prediction_intervals**(*self, X: pandas.DataFrame, y: pandas.Series = None, coverage: List[float] = None, predictions: pandas.Series = None*) → Dict[str, pandas.Series]

Find the prediction intervals using the fitted ARIMARegressor.

**Parameters**

- **X** (*pd.DataFrame*) – Data of shape [n_samples, n_features].
- **y** (*pd.Series*) – Target data. Optional.
- **coverage** (*list[float]*) – A list of floats between the values 0 and 1 that the upper and lower bounds of the prediction interval should be calculated for.
- **predictions** (*pd.Series*) – Not used for ARIMA regressor.

**Returns** Prediction intervals, keys are in the format {coverage}_lower or {coverage}_upper.

**Return type** dict

**static load**(*file_path*)

Loads component at file path.

**Parameters** **file_path** (*str*) – Location to load file.

**Returns** ComponentBase object

**needs_fitting**(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need to be fit or whose fit methods do nothing.

**Returns** True.

**property parameters**(*self*)

Returns the parameters which were used to initialize the component.

**predict**(*self*, *X: pandas.DataFrame*, *y: Optional[pandas.Series] = None*) → pandas.Series

Make predictions using fitted ARIMA regressor.

**Parameters**

- **X** (*pd.DataFrame*) – Data of shape [n_samples, n_features].
- **y** (*pd.Series*) – Target data.

**Returns** Predicted values.

**Return type** pd.Series

**Raises** **ValueError** – If X was passed to *fit* but not passed in *predict*.

**predict_proba**(*self*, *X: pandas.DataFrame*) → pandas.Series

Make probability estimates for labels.

**Parameters** **X** (*pd.DataFrame*) – Features.

**Returns** Probability estimates.

**Return type** pd.Series

**Raises** **MethodPropertyNotFoundError** – If estimator does not have a predict_proba method or a component_obj that implements predict_proba.

**save**(*self*, *file_path*, *pickle_protocol=cloudpickle.DEFAULT_PROTOCOL*)

Saves component at file path.

**Parameters**

- **file_path** (*str*) – Location to save file.
- **pickle_protocol** (*int*) – The pickle data stream format.

**update_parameters**(*self*, *update_dict*, *reset_fit=True*)

Updates the parameter dictionary of the component.

**Parameters**

- **update_dict** (*dict*) – A dict of parameters to update.
- **reset_fit** (*bool, optional*) – If True, will set *_is_fitted* to False.

**class** evalml.pipelines.components.**BaselineClassifier**(*strategy='mode'*, *random_seed=0*, ***kwargs*)

Classifier that predicts using the specified strategy.

This is useful as a simple baseline classifier to compare with other classifiers.

**Parameters**

- **strategy** (*str*) – Method used to predict. Valid options are "mode", "random" and "random_weighted". Defaults to "mode".
- **random_seed** (*int*) – Seed for the random number generator. Defaults to 0.

**Attributes**

| | |
|---|---|
| **hyper-parameter_ranges** | {} |
| **model_family** | ModelFamily.BASELINE |
| **modifies_features** | True |
| **modifies_target** | False |
| **name** | Baseline Classifier |
| **supported_problem_types** | [ProblemTypes.BINARY, ProblemTypes.MULTICLASS] |
| **training_only** | False |

**Methods**

| | |
|---|---|
| *classes_* | Returns class labels. Will return None before fitting. |
| *clone* | Constructs a new component with the same parameters and random state. |
| *default_parameters* | Returns the default parameters for this component. |
| *describe* | Describe a component and its parameters. |
| *feature_importance* | Returns importance associated with each feature. Since baseline classifiers do not use input features to calculate predictions, returns an array of zeroes. |
| *fit* | Fits baseline classifier component to data. |
| *get_prediction_intervals* | Find the prediction intervals using the fitted regressor. |
| *load* | Loads component at file path. |
| *needs_fitting* | Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances. |
| *parameters* | Returns the parameters which were used to initialize the component. |
| *predict* | Make predictions using the baseline classification strategy. |
| *predict_proba* | Make prediction probabilities using the baseline classification strategy. |
| *save* | Saves component at file path. |
| *update_parameters* | Updates the parameter dictionary of the component. |

**property classes_**(*self*)

Returns class labels. Will return None before fitting.

**Returns** Class names

**Return type** list[str] or list(float)

**clone**(*self*)

Constructs a new component with the same parameters and random state.

**Returns** A new instance of this component with identical parameters and random state.

**default_parameters**(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

**Returns** Default parameters for this component.

**Return type** dict

**describe**(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

**Parameters**

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description as dictionary in the format {"name": name, "parameters": parameters}

**Returns** Returns dictionary if return_dict is True, else None.

**Return type** None or dict

**property feature_importance**(*self*)

Returns importance associated with each feature. Since baseline classifiers do not use input features to calculate predictions, returns an array of zeroes.

**Returns** An array of zeroes

**Return type** pd.Series

**fit**(*self*, *X*, *y=None*)

Fits baseline classifier component to data.

**Parameters**

- **X** (*pd.DataFrame*) – The input training data of shape [n_samples, n_features].
- **y** (*pd.Series*) – The target training data of length [n_samples].

**Returns** self

**Raises** **ValueError** – If y is None.

**get_prediction_intervals**(*self*, *X: pandas.DataFrame*, *y: Optional[pandas.Series] = None*, *coverage: List[float] = None*, *predictions: pandas.Series = None*) → Dict[str, pandas.Series]

Find the prediction intervals using the fitted regressor.

This function takes the predictions of the fitted estimator and calculates the rolling standard deviation across all predictions using a window size of 5. The lower and upper predictions are determined by taking the percent point (quantile) function of the lower tail probability at each bound multiplied by the rolling standard deviation.

**Parameters**

- **X** (*pd.DataFrame*) – Data of shape [n_samples, n_features].
- **y** (*pd.Series*) – Target data. Ignored.
- **coverage** (*list[float]*) – A list of floats between the values 0 and 1 that the upper and lower bounds of the prediction interval should be calculated for.
- **predictions** (*pd.Series*) – Optional list of predictions to use. If None, will generate predictions using *X*.

**Returns** Prediction intervals, keys are in the format {coverage}_lower or {coverage}_upper.

**Return type** dict

**Raises** **MethodPropertyNotFoundError** – If the estimator does not support Time Series Regression as a problem type.

**static load**(*file_path*)

Loads component at file path.

**Parameters** **file_path** (*str*) – Location to load file.

**Returns** ComponentBase object

**needs_fitting**(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need to be fit or whose fit methods do nothing.

**Returns** True.

**property parameters**(*self*)

Returns the parameters which were used to initialize the component.

**predict**(*self*, *X*)

Make predictions using the baseline classification strategy.

**Parameters** **X** (*pd.DataFrame*) – Data of shape [n_samples, n_features].

**Returns** Predicted values.

**Return type** pd.Series

**predict_proba**(*self*, *X*)

Make prediction probabilities using the baseline classification strategy.

**Parameters** **X** (*pd.DataFrame*) – Data of shape [n_samples, n_features].

**Returns** Predicted probability values.

**Return type** pd.DataFrame

**save**(*self*, *file_path*, *pickle_protocol=cloudpickle.DEFAULT_PROTOCOL*)

Saves component at file path.

**Parameters**

- **file_path** (*str*) – Location to save file.
- **pickle_protocol** (*int*) – The pickle data stream format.

**update_parameters**(*self*, *update_dict*, *reset_fit=True*)

Updates the parameter dictionary of the component.

**Parameters**

- **update_dict** (*dict*) – A dict of parameters to update.
- **reset_fit** (*bool, optional*) – If True, will set *_is_fitted* to False.

**class** evalml.pipelines.components.**BaselineRegressor**(*strategy='mean'*, *random_seed=0*, ***kwargs*)

Baseline regressor that uses a simple strategy to make predictions. This is useful as a simple baseline regressor to compare with other regressors.

**Parameters**

- **strategy** (*str*) – Method used to predict. Valid options are "mean", "median". Defaults to "mean".

- **random_seed** (*int*) – Seed for the random number generator. Defaults to 0.

**Attributes**

| | |
|---|---|
| **hyper-parameter_ranges** | {} |
| **model_family** | ModelFamily.BASELINE |
| **modifies_features** | True |
| **modifies_target** | False |
| **name** | Baseline Regressor |
| **supported_problem_types** | [ ProblemTypes.REGRESSION, ProblemTypes.TIME_SERIES_REGRESSION,] |
| **training_only** | False |

**Methods**

| | |
|---|---|
| *clone* | Constructs a new component with the same parameters and random state. |
| *default_parameters* | Returns the default parameters for this component. |
| *describe* | Describe a component and its parameters. |
| *feature_importance* | Returns importance associated with each feature. Since baseline regressors do not use input features to calculate predictions, returns an array of zeroes. |
| *fit* | Fits baseline regression component to data. |
| *get_prediction_intervals* | Find the prediction intervals using the fitted regressor. |
| *load* | Loads component at file path. |
| *needs_fitting* | Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances. |
| *parameters* | Returns the parameters which were used to initialize the component. |
| *predict* | Make predictions using the baseline regression strategy. |
| *predict_proba* | Make probability estimates for labels. |
| *save* | Saves component at file path. |
| *update_parameters* | Updates the parameter dictionary of the component. |

**clone**(*self*)

Constructs a new component with the same parameters and random state.

**Returns** A new instance of this component with identical parameters and random state.

**default_parameters**(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

**Returns** Default parameters for this component.

**Return type** dict

**describe**(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

**Parameters**

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description as dictionary in the format {"name": name, "parameters": parameters}

**Returns** Returns dictionary if return_dict is True, else None.

**Return type** None or dict

**property feature_importance**(*self*)

Returns importance associated with each feature. Since baseline regressors do not use input features to calculate predictions, returns an array of zeroes.

**Returns** An array of zeroes.

**Return type** np.ndarray (float)

**fit**(*self*, *X*, *y=None*)

Fits baseline regression component to data.

**Parameters**

- **X** (*pd.DataFrame*) – The input training data of shape [n_samples, n_features].
- **y** (*pd.Series*) – The target training data of length [n_samples].

**Returns** self

**Raises** **ValueError** – If input y is None.

**get_prediction_intervals**(*self*, *X: pandas.DataFrame*, *y: Optional[pandas.Series] = None*, *coverage: List[float] = None*, *predictions: pandas.Series = None*) → Dict[str, pandas.Series]

Find the prediction intervals using the fitted regressor.

This function takes the predictions of the fitted estimator and calculates the rolling standard deviation across all predictions using a window size of 5. The lower and upper predictions are determined by taking the percent point (quantile) function of the lower tail probability at each bound multiplied by the rolling standard deviation.

**Parameters**

- **X** (*pd.DataFrame*) – Data of shape [n_samples, n_features].
- **y** (*pd.Series*) – Target data. Ignored.
- **coverage** (*list[float]*) – A list of floats between the values 0 and 1 that the upper and lower bounds of the prediction interval should be calculated for.
- **predictions** (*pd.Series*) – Optional list of predictions to use. If None, will generate predictions using *X*.

**Returns** Prediction intervals, keys are in the format {coverage}_lower or {coverage}_upper.

**Return type** dict

**Raises** **MethodPropertyNotFoundError** – If the estimator does not support Time Series Regression as a problem type.

**static load**(*file_path*)

Loads component at file path.

**Parameters** **file_path** (*str*) – Location to load file.

**Returns** ComponentBase object

**needs_fitting**(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need to be fit or whose fit methods do nothing.

**Returns** True.

**property parameters**(*self*)

Returns the parameters which were used to initialize the component.

**predict**(*self*, *X*)

Make predictions using the baseline regression strategy.

**Parameters** **X** (*pd.DataFrame*) – Data of shape [n_samples, n_features].

**Returns** Predicted values.

**Return type** pd.Series

**predict_proba**(*self*, *X: pandas.DataFrame*) → pandas.Series

Make probability estimates for labels.

**Parameters** **X** (*pd.DataFrame*) – Features.

**Returns** Probability estimates.

**Return type** pd.Series

**Raises** **MethodPropertyNotFoundError** – If estimator does not have a predict_proba method or a component_obj that implements predict_proba.

**save**(*self*, *file_path*, *pickle_protocol=cloudpickle.DEFAULT_PROTOCOL*)

Saves component at file path.

**Parameters**

- **file_path** (*str*) – Location to save file.
- **pickle_protocol** (*int*) – The pickle data stream format.

**update_parameters**(*self*, *update_dict*, *reset_fit=True*)

Updates the parameter dictionary of the component.

**Parameters**

- **update_dict** (*dict*) – A dict of parameters to update.
- **reset_fit** (*bool, optional*) – If True, will set *_is_fitted* to False.

**class** evalml.pipelines.components.**CatBoostClassifier**(*n_estimators=10*, *eta=0.03*, *max_depth=6*, *bootstrap_type=None*, *silent=True*, *allow_writing_files=False*, *random_seed=0*, *n_jobs=- 1*, ***kwargs*)

CatBoost Classifier, a classifier that uses gradient-boosting on decision trees. CatBoost is an open-source library and natively supports categorical features.

For more information, check out https://catboost.ai/

**Parameters**

- **n_estimators** (*float*) – The maximum number of trees to build. Defaults to 10.
- **eta** (*float*) – The learning rate. Defaults to 0.03.
- **max_depth** (*int*) – The maximum tree depth for base learners. Defaults to 6.
- **bootstrap_type** (*string*) – Defines the method for sampling the weights of objects. Available methods are 'Bayesian', 'Bernoulli', 'MVS'. Defaults to None.
- **silent** (*boolean*) – Whether to use the "silent" logging mode. Defaults to True.
- **allow_writing_files** (*boolean*) – Whether to allow writing snapshot files while training. Defaults to False.
- **n_jobs** (*int or None*) – Number of jobs to run in parallel. -1 uses all processes. Defaults to -1.
- **random_seed** (*int*) – Seed for the random number generator. Defaults to 0.

**Attributes**

| | |
|---|---|
| **hyperparameter_ranges** | { "n_estimators": Integer(4, 100), "eta": Real(0.000001, 1), "max_depth": Integer(4, 10),} |
| **model_family** | ModelFamily.CATBOOST |
| **modifies_features** | True |
| **modifies_target** | False |
| **name** | CatBoost Classifier |
| **supported_problem_types** | [ ProblemTypes.BINARY, ProblemTypes.MULTICLASS, ProblemTypes.TIME_SERIES_BINARY, ProblemTypes.TIME_SERIES_MULTICLASS,] |
| **training_only** | False |

**Methods**

| | |
|---|---|
| *clone* | Constructs a new component with the same parameters and random state. |
| *default_parameters* | Returns the default parameters for this component. |
| *describe* | Describe a component and its parameters. |
| *feature_importance* | Feature importance of fitted CatBoost classifier. |
| *fit* | Fits CatBoost classifier component to data. |
| *get_prediction_intervals* | Find the prediction intervals using the fitted regressor. |
| *load* | Loads component at file path. |
| *needs_fitting* | Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances. |
| *parameters* | Returns the parameters which were used to initialize the component. |
| *predict* | Make predictions using the fitted CatBoost classifier. |
| *predict_proba* | Make prediction probabilities using the fitted CatBoost classifier. |
| *save* | Saves component at file path. |
| *update_parameters* | Updates the parameter dictionary of the component. |

**clone**(*self*)

Constructs a new component with the same parameters and random state.

**Returns** A new instance of this component with identical parameters and random state.

**default_parameters**(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

**Returns** Default parameters for this component.

**Return type** dict

**describe**(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

**Parameters**

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description as dictionary in the format {"name": name, "parameters": parameters}

**Returns** Returns dictionary if return_dict is True, else None.

**Return type** None or dict

**property feature_importance**(*self*)

Feature importance of fitted CatBoost classifier.

**fit**(*self*, *X*, *y=None*)

Fits CatBoost classifier component to data.

**Parameters**

- **X** (*pd.DataFrame*) – The input training data of shape [n_samples, n_features].
- **y** (*pd.Series*) – The target training data of length [n_samples].

**Returns** self

**get_prediction_intervals**(*self*, *X: pandas.DataFrame*, *y: Optional[pandas.Series] = None*, *coverage: List[float] = None*, *predictions: pandas.Series = None*) → Dict[str, pandas.Series]

Find the prediction intervals using the fitted regressor.

This function takes the predictions of the fitted estimator and calculates the rolling standard deviation across all predictions using a window size of 5. The lower and upper predictions are determined by taking the percent point (quantile) function of the lower tail probability at each bound multiplied by the rolling standard deviation.

**Parameters**

- **X** (*pd.DataFrame*) – Data of shape [n_samples, n_features].
- **y** (*pd.Series*) – Target data. Ignored.
- **coverage** (*list[float]*) – A list of floats between the values 0 and 1 that the upper and lower bounds of the prediction interval should be calculated for.
- **predictions** (*pd.Series*) – Optional list of predictions to use. If None, will generate predictions using *X*.

**Returns** Prediction intervals, keys are in the format {coverage}_lower or {coverage}_upper.

**Return type** dict

**Raises** **MethodPropertyNotFoundError** – If the estimator does not support Time Series Regression as a problem type.

**static load**(*file_path*)

Loads component at file path.

**Parameters** **file_path** (*str*) – Location to load file.

**Returns** ComponentBase object

**needs_fitting**(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need to be fit or whose fit methods do nothing.

**Returns** True.

**property parameters**(*self*)

Returns the parameters which were used to initialize the component.

**predict**(*self*, *X*)

Make predictions using the fitted CatBoost classifier.

**Parameters** **X** (*pd.DataFrame*) – Data of shape [n_samples, n_features].

**Returns** Predicted values.

**Return type** pd.Series

**predict_proba**(*self*, *X*)

Make prediction probabilities using the fitted CatBoost classifier.

**Parameters** **X** (*pd.DataFrame*) – Data of shape [n_samples, n_features].

**Returns** Predicted probability values.

**Return type** pd.DataFrame

**save**(*self*, *file_path*, *pickle_protocol=cloudpickle.DEFAULT_PROTOCOL*)

Saves component at file path.

**Parameters**

- **file_path** (*str*) – Location to save file.
- **pickle_protocol** (*int*) – The pickle data stream format.

**update_parameters**(*self*, *update_dict*, *reset_fit=True*)

Updates the parameter dictionary of the component.

**Parameters**

- **update_dict** (*dict*) – A dict of parameters to update.
- **reset_fit** (*bool, optional*) – If True, will set *_is_fitted* to False.

class evalml.pipelines.components.**CatBoostRegressor**(*n_estimators=10*, *eta=0.03*, *max_depth=6*, *bootstrap_type=None*, *silent=False*, *allow_writing_files=False*, *random_seed=0*, *n_jobs=- 1*, ***kwargs*)

CatBoost Regressor, a regressor that uses gradient-boosting on decision trees. CatBoost is an open-source library and natively supports categorical features.

For more information, check out https://catboost.ai/

**Parameters**

- **n_estimators** (*float*) – The maximum number of trees to build. Defaults to 10.
- **eta** (*float*) – The learning rate. Defaults to 0.03.
- **max_depth** (*int*) – The maximum tree depth for base learners. Defaults to 6.
- **bootstrap_type** (*string*) – Defines the method for sampling the weights of objects. Available methods are 'Bayesian', 'Bernoulli', 'MVS'. Defaults to None.
- **silent** (*boolean*) – Whether to use the "silent" logging mode. Defaults to True.
- **allow_writing_files** (*boolean*) – Whether to allow writing snapshot files while training. Defaults to False.
- **n_jobs** (*int or None*) – Number of jobs to run in parallel. -1 uses all processes. Defaults to -1.
- **random_seed** (*int*) – Seed for the random number generator. Defaults to 0.

**Attributes**

| | |
|---|---|
| **hyper-parameter_ranges** | { "n_estimators": Integer(4, 100), "eta": Real(0.000001, 1), "max_depth": Integer(4, 10),} |
| **model_family** | ModelFamily.CATBOOST |
| **modifies_features** | True |
| **modifies_target** | False |
| **name** | CatBoost Regressor |
| **supported_problem_types** | [ ProblemTypes.REGRESSION, ProblemTypes.TIME_SERIES_REGRESSION,] |
| **training_only** | False |

**Methods**

| | |
|---|---|
| *clone* | Constructs a new component with the same parameters and random state. |
| *default_parameters* | Returns the default parameters for this component. |
| *describe* | Describe a component and its parameters. |
| *feature_importance* | Feature importance of fitted CatBoost regressor. |
| *fit* | Fits CatBoost regressor component to data. |
| *get_prediction_intervals* | Find the prediction intervals using the fitted regressor. |
| *load* | Loads component at file path. |
| *needs_fitting* | Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances. |
| *parameters* | Returns the parameters which were used to initialize the component. |
| *predict* | Make predictions using the fitted CatBoost regressor. |
| *predict_proba* | Make probability estimates for labels. |
| *save* | Saves component at file path. |
| *update_parameters* | Updates the parameter dictionary of the component. |

**clone**(*self*)

Constructs a new component with the same parameters and random state.

**Returns** A new instance of this component with identical parameters and random state.

**default_parameters**(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

**Returns** Default parameters for this component.

**Return type** dict

**describe**(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

**Parameters**

- **print_name** (*bool, optional*) – whether to print name of component

- **return_dict** (*bool, optional*) – whether to return description as dictionary in the format {"name": name, "parameters": parameters}

**Returns** Returns dictionary if return_dict is True, else None.

**Return type** None or dict

**property feature_importance**(*self*)

Feature importance of fitted CatBoost regressor.

**fit**(*self*, *X*, *y=None*)

Fits CatBoost regressor component to data.

**Parameters**

- **X** (*pd.DataFrame*) – The input training data of shape [n_samples, n_features].
- **y** (*pd.Series*) – The target training data of length [n_samples].

**Returns** self

**get_prediction_intervals**(*self*, *X: pandas.DataFrame*, *y: Optional[pandas.Series] = None*, *coverage: List[float] = None*, *predictions: pandas.Series = None*) → Dict[str, pandas.Series]

Find the prediction intervals using the fitted regressor.

This function takes the predictions of the fitted estimator and calculates the rolling standard deviation across all predictions using a window size of 5. The lower and upper predictions are determined by taking the percent point (quantile) function of the lower tail probability at each bound multiplied by the rolling standard deviation.

**Parameters**

- **X** (*pd.DataFrame*) – Data of shape [n_samples, n_features].
- **y** (*pd.Series*) – Target data. Ignored.
- **coverage** (*list[float]*) – A list of floats between the values 0 and 1 that the upper and lower bounds of the prediction interval should be calculated for.
- **predictions** (*pd.Series*) – Optional list of predictions to use. If None, will generate predictions using *X*.

**Returns** Prediction intervals, keys are in the format {coverage}_lower or {coverage}_upper.

**Return type** dict

**Raises** **MethodPropertyNotFoundError** – If the estimator does not support Time Series Regression as a problem type.

**static load**(*file_path*)

Loads component at file path.

**Parameters** **file_path** (*str*) – Location to load file.

**Returns** ComponentBase object

**needs_fitting**(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need to be fit or whose fit methods do nothing.

**Returns** True.

**property parameters**(*self*)

Returns the parameters which were used to initialize the component.

**predict**(*self*, *X*)

Make predictions using the fitted CatBoost regressor.

**Parameters** **X** (*pd.DataFrame*) – Data of shape [n_samples, n_features].

**Returns** Predicted values.

**Return type** pd.DataFrame

**predict_proba**(*self*, *X: pandas.DataFrame*) → pandas.Series

Make probability estimates for labels.

**Parameters** **X** (*pd.DataFrame*) – Features.

**Returns** Probability estimates.

**Return type** pd.Series

**Raises** **MethodPropertyNotFoundError** – If estimator does not have a predict_proba method or a component_obj that implements predict_proba.

**save**(*self*, *file_path*, *pickle_protocol=cloudpickle.DEFAULT_PROTOCOL*)

Saves component at file path.

**Parameters**

- **file_path** (*str*) – Location to save file.
- **pickle_protocol** (*int*) – The pickle data stream format.

**update_parameters**(*self*, *update_dict*, *reset_fit=True*)

Updates the parameter dictionary of the component.

**Parameters**

- **update_dict** (*dict*) – A dict of parameters to update.
- **reset_fit** (*bool, optional*) – If True, will set *_is_fitted* to False.

**class** evalml.pipelines.components.**ComponentBase**(*parameters=None*, *component_obj=None*, *random_seed=0*, ***kwargs*)

Base class for all components.

**Parameters**

- **parameters** (*dict*) – Dictionary of parameters for the component. Defaults to None.
- **component_obj** (*obj*) – Third-party objects useful in component implementation. Defaults to None.
- **random_seed** (*int*) – Seed for the random number generator. Defaults to 0.

**Methods**

| | |
|---|---|
| clone | Constructs a new component with the same parameters and random state. |
| default_parameters | Returns the default parameters for this component. |
| describe | Describe a component and its parameters. |
| fit | Fits component to data. |
| load | Loads component at file path. |
| modifies_features | Returns whether this component modifies (subsets or transforms) the features variable during transform. |
| modifies_target | Returns whether this component modifies (subsets or transforms) the target variable during transform. |
| name | Returns string name of this component. |
| needs_fitting | Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances. |
| parameters | Returns the parameters which were used to initialize the component. |
| save | Saves component at file path. |
| training_only | Returns whether or not this component should be evaluated during training-time only, or during both training and prediction time. |
| update_parameters | Updates the parameter dictionary of the component. |

**clone**(*self*)

Constructs a new component with the same parameters and random state.

**Returns** A new instance of this component with identical parameters and random state.

**default_parameters**(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

**Returns** Default parameters for this component.

**Return type** dict

**describe**(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

**Parameters**

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description as dictionary in the format {"name": name, "parameters": parameters}

**Returns** Returns dictionary if return_dict is True, else None.

**Return type** None or dict

**fit**(*self*, *X*, *y=None*)

Fits component to data.

**Parameters**

- **X** (*pd.DataFrame*) – The input training data of shape [n_samples, n_features]
- **y** (*pd.Series, optional*) – The target training data of length [n_samples]

**Returns** self

**Raises** **MethodPropertyNotFoundError** – If component does not have a fit method or a component_obj that implements fit.

**static load**(*file_path*)

Loads component at file path.

**Parameters** **file_path** (*str*) – Location to load file.

**Returns** ComponentBase object

**property modifies_features**(*cls*)

Returns whether this component modifies (subsets or transforms) the features variable during transform.

For Estimator objects, this attribute determines if the return value from *predict* or *predict_proba* should be used as features or targets.

**property modifies_target**(*cls*)

Returns whether this component modifies (subsets or transforms) the target variable during transform.

For Estimator objects, this attribute determines if the return value from *predict* or *predict_proba* should be used as features or targets.

**property name**(*cls*)

Returns string name of this component.

**needs_fitting**(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need to be fit or whose fit methods do nothing.

**Returns** True.

**property parameters**(*self*)

Returns the parameters which were used to initialize the component.

**save**(*self*, *file_path*, *pickle_protocol=cloudpickle.DEFAULT_PROTOCOL*)

Saves component at file path.

**Parameters**

- **file_path** (*str*) – Location to save file.
- **pickle_protocol** (*int*) – The pickle data stream format.

**property training_only**(*cls*)

Returns whether or not this component should be evaluated during training-time only, or during both training and prediction time.

**update_parameters**(*self*, *update_dict*, *reset_fit=True*)

Updates the parameter dictionary of the component.

**Parameters**

- **update_dict** (*dict*) – A dict of parameters to update.
- **reset_fit** (*bool, optional*) – If True, will set *_is_fitted* to False.

**class** evalml.pipelines.components.**ComponentBaseMeta**

Metaclass that overrides creating a new component by wrapping methods with validators and setters.

**Attributes**

| | |
|---|---|
| **FIT_METHODS** | ['fit', 'fit_transform'] |
| **METHODS_TO_CHECK** | ['predict', 'predict_proba', 'transform', 'inverse_transform', 'get_trend_dataframe'] |
| **PROPERTIES_TO_CHECK** | ['feature_importance'] |

**Methods**

| | |
|---|---|
| *check_for_fit* | *check_for_fit* wraps a method that validates if *self._is_fitted* is *True*. |
| *register* | Register a virtual subclass of an ABC. |
| *set_fit* | Wrapper for the fit method. |

**classmethod check_for_fit**(*cls*, *method*)

*check_for_fit* wraps a method that validates if *self._is_fitted* is *True*.

It raises an exception if *False* and calls and returns the wrapped method if *True*.

**Parameters** **method** (*callable*) – Method to wrap.

**Returns** The wrapped method.

**Raises** **ComponentNotYetFittedError** – If component is not yet fitted.

**register**(*cls*, *subclass*)

Register a virtual subclass of an ABC.

Returns the subclass, to allow usage as a class decorator.

**classmethod set_fit**(*cls*, *method*)

Wrapper for the fit method.

**class** evalml.pipelines.components.**DateTimeFeaturizer**(*features_to_extract=None*, *encode_as_categories=False*, *time_index=None*, *random_seed=0*, ***kwargs*)

Transformer that can automatically extract features from datetime columns.

**Parameters**

- **features_to_extract** (*list*) – List of features to extract. Valid options include "year", "month", "day_of_week", "hour". Defaults to None.
- **encode_as_categories** (*bool*) – Whether day-of-week and month features should be encoded as pandas "category" dtype. This allows OneHotEncoders to encode these features. Defaults to False.
- **time_index** (*str*) – Name of the column containing the datetime information used to order the data. Ignored.
- **random_seed** (*int*) – Seed for the random number generator. Defaults to 0.

**Attributes**

| | |
|---|---|
| **hyper-parameter_ranges** | {} |
| **modifies_features** | True |
| **modifies_target** | False |
| **name** | DateTime Featurizer |
| **training_only** | False |

**Methods**

| | |
|---|---|
| *clone* | Constructs a new component with the same parameters and random state. |
| *default_parameters* | Returns the default parameters for this component. |
| *describe* | Describe a component and its parameters. |
| *fit* | Fit the datetime featurizer component. |
| *fit_transform* | Fits on X and transforms X. |
| *get_feature_names* | Gets the categories of each datetime feature. |
| *load* | Loads component at file path. |
| *needs_fitting* | Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances. |
| *parameters* | Returns the parameters which were used to initialize the component. |
| *save* | Saves component at file path. |
| *transform* | Transforms data X by creating new features using existing DateTime columns, and then dropping those DateTime columns. |
| *update_parameters* | Updates the parameter dictionary of the component. |

**clone**(*self*)

Constructs a new component with the same parameters and random state.

**Returns** A new instance of this component with identical parameters and random state.

**default_parameters**(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

**Returns** Default parameters for this component.

**Return type** dict

**describe**(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

**Parameters**

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description as dictionary in the format {"name": name, "parameters": parameters}

**Returns** Returns dictionary if return_dict is True, else None.

**Return type** None or dict

`fit`(*self*, *X*, *y=None*)

Fit the datetime featurizer component.

**Parameters**

- **X** (*pd.DataFrame*) – Input features.
- **y** (*pd.Series, optional*) – Target data. Ignored.

**Returns** self

`fit_transform`(*self*, *X*, *y=None*)

Fits on X and transforms X.

**Parameters**

- **X** (*pd.DataFrame*) – Data to fit and transform.
- **y** (*pd.Series*) – Target data.

**Returns** Transformed X.

**Return type** pd.DataFrame

**Raises** **MethodPropertyNotFoundError** – If transformer does not have a transform method or a component_obj that implements transform.

`get_feature_names`(*self*)

Gets the categories of each datetime feature.

**Returns**

**Dictionary, where each key-value pair is a column name and a dictionary** mapping the unique feature values to their integer encoding.

**Return type** dict

`static load`(*file_path*)

Loads component at file path.

**Parameters** **file_path** (*str*) – Location to load file.

**Returns** ComponentBase object

`needs_fitting`(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need to be fit or whose fit methods do nothing.

**Returns** True.

`property parameters`(*self*)

Returns the parameters which were used to initialize the component.

`save`(*self*, *file_path*, *pickle_protocol=cloudpickle.DEFAULT_PROTOCOL*)

Saves component at file path.

**Parameters**

- **file_path** (*str*) – Location to save file.

- **pickle_protocol** (*int*) – The pickle data stream format.

**transform**(*self*, *X*, *y=None*)

Transforms data X by creating new features using existing DateTime columns, and then dropping those DateTime columns.

**Parameters**

- **X** (*pd.DataFrame*) – Input features.
- **y** (*pd.Series, optional*) – Ignored.

**Returns** Transformed X

**Return type** pd.DataFrame

**update_parameters**(*self*, *update_dict*, *reset_fit=True*)

Updates the parameter dictionary of the component.

**Parameters**

- **update_dict** (*dict*) – A dict of parameters to update.
- **reset_fit** (*bool, optional*) – If True, will set *_is_fitted* to False.

**class** evalml.pipelines.components.**DecisionTreeClassifier**(*criterion='gini'*, *max_features='auto'*, *max_depth=6*, *min_samples_split=2*, *min_weight_fraction_leaf=0.0*, *random_seed=0*, ***kwargs*)

Decision Tree Classifier.

**Parameters**

- **criterion** (*{"gini", "entropy"}*) – The function to measure the quality of a split. Supported criteria are "gini" for the Gini impurity and "entropy" for the information gain. Defaults to "gini".
- **max_features** (*int, float or {"auto", "sqrt", "log2"}*) – The number of features to consider when looking for the best split:
  - If int, then consider max_features features at each split.
  - If float, then max_features is a fraction and int(max_features * n_features) features are considered at each split.
  - If "auto", then max_features=sqrt(n_features).
  - If "sqrt", then max_features=sqrt(n_features).
  - If "log2", then max_features=log2(n_features).
  - If None, then max_features = n_features.

  The search for a split does not stop until at least one valid partition of the node samples is found, even if it requires to effectively inspect more than max_features features. Defaults to "auto".
- **max_depth** (*int*) – The maximum depth of the tree. Defaults to 6.
- **min_samples_split** (*int or float*) – The minimum number of samples required to split an internal node:
  - If int, then consider min_samples_split as the minimum number.
  - If float, then min_samples_split is a fraction and ceil(min_samples_split * n_samples) are the minimum number of samples for each split.

Defaults to 2.

- **min_weight_fraction_leaf** (*float*) – The minimum weighted fraction of the sum total of weights (of all the input samples) required to be at a leaf node. Defaults to 0.0.
- **random_seed** (*int*) – Seed for the random number generator. Defaults to 0.

**Attributes**

| | |
|---|---|
| **hyper-parameter_ranges** | { "criterion": ["gini", "entropy"], "max_features": ["auto", "sqrt", "log2"], "max_depth": Integer(4, 10),} |
| **model_family** | ModelFamily.DECISION_TREE |
| **modifies_features** | True |
| **modifies_target** | False |
| **name** | Decision Tree Classifier |
| **supported_problem_types** | [ ProblemTypes.BINARY, ProblemTypes.MULTICLASS, ProblemTypes.TIME_SERIES_BINARY, ProblemTypes.TIME_SERIES_MULTICLASS,] |
| **training_only** | False |

**Methods**

| | |
|---|---|
| *clone* | Constructs a new component with the same parameters and random state. |
| *default_parameters* | Returns the default parameters for this component. |
| *describe* | Describe a component and its parameters. |
| *feature_importance* | Returns importance associated with each feature. |
| *fit* | Fits estimator to data. |
| *get_prediction_intervals* | Find the prediction intervals using the fitted regressor. |
| *load* | Loads component at file path. |
| *needs_fitting* | Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances. |
| *parameters* | Returns the parameters which were used to initialize the component. |
| *predict* | Make predictions using selected features. |
| *predict_proba* | Make probability estimates for labels. |
| *save* | Saves component at file path. |
| *update_parameters* | Updates the parameter dictionary of the component. |

**clone**(*self*)

Constructs a new component with the same parameters and random state.

**Returns** A new instance of this component with identical parameters and random state.

**default_parameters**(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

**Returns** Default parameters for this component.

**Return type** dict

**describe**(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

**Parameters**

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description as dictionary in the format {"name": name, "parameters": parameters}

**Returns** Returns dictionary if return_dict is True, else None.

**Return type** None or dict

**property feature_importance**(*self*) → pandas.Series

Returns importance associated with each feature.

**Returns** Importance associated with each feature.

**Return type** np.ndarray

**Raises** **MethodPropertyNotFoundError** – If estimator does not have a feature_importance method or a component_obj that implements feature_importance.

**fit**(*self*, *X: pandas.DataFrame*, *y: Optional[pandas.Series] = None*)

Fits estimator to data.

**Parameters**

- **X** (*pd.DataFrame*) – The input training data of shape [n_samples, n_features].
- **y** (*pd.Series, optional*) – The target training data of length [n_samples].

**Returns** self

**get_prediction_intervals**(*self*, *X: pandas.DataFrame*, *y: Optional[pandas.Series] = None*, *coverage: List[float] = None*, *predictions: pandas.Series = None*) → Dict[str, pandas.Series]

Find the prediction intervals using the fitted regressor.

This function takes the predictions of the fitted estimator and calculates the rolling standard deviation across all predictions using a window size of 5. The lower and upper predictions are determined by taking the percent point (quantile) function of the lower tail probability at each bound multiplied by the rolling standard deviation.

**Parameters**

- **X** (*pd.DataFrame*) – Data of shape [n_samples, n_features].
- **y** (*pd.Series*) – Target data. Ignored.
- **coverage** (*list[float]*) – A list of floats between the values 0 and 1 that the upper and lower bounds of the prediction interval should be calculated for.
- **predictions** (*pd.Series*) – Optional list of predictions to use. If None, will generate predictions using *X*.

**Returns** Prediction intervals, keys are in the format {coverage}_lower or {coverage}_upper.

**Return type** dict

**Raises** **MethodPropertyNotFoundError** – If the estimator does not support Time Series Regression as a problem type.

**static load**(*file_path*)

Loads component at file path.

**Parameters** **file_path** (*str*) – Location to load file.

**Returns** ComponentBase object

**needs_fitting**(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need to be fit or whose fit methods do nothing.

**Returns** True.

**property parameters**(*self*)

Returns the parameters which were used to initialize the component.

**predict**(*self*, *X: pandas.DataFrame*) → pandas.Series

Make predictions using selected features.

**Parameters** **X** (*pd.DataFrame*) – Data of shape [n_samples, n_features].

**Returns** Predicted values.

**Return type** pd.Series

**Raises** **MethodPropertyNotFoundError** – If estimator does not have a predict method or a component_obj that implements predict.

**predict_proba**(*self*, *X: pandas.DataFrame*) → pandas.Series

Make probability estimates for labels.

**Parameters** **X** (*pd.DataFrame*) – Features.

**Returns** Probability estimates.

**Return type** pd.Series

**Raises** **MethodPropertyNotFoundError** – If estimator does not have a predict_proba method or a component_obj that implements predict_proba.

**save**(*self*, *file_path*, *pickle_protocol=cloudpickle.DEFAULT_PROTOCOL*)

Saves component at file path.

**Parameters**

- **file_path** (*str*) – Location to save file.
- **pickle_protocol** (*int*) – The pickle data stream format.

**update_parameters**(*self*, *update_dict*, *reset_fit=True*)

Updates the parameter dictionary of the component.

**Parameters**

- **update_dict** (*dict*) – A dict of parameters to update.
- **reset_fit** (*bool, optional*) – If True, will set *_is_fitted* to False.

**class** evalml.pipelines.components.**DecisionTreeRegressor**(*criterion='squared_error', max_features='auto', max_depth=6, min_samples_split=2, min_weight_fraction_leaf=0.0, random_seed=0, **kwargs*)

Decision Tree Regressor.

**Parameters**

- **criterion** (*{"squared_error", "friedman_mse", "absolute_error", "poisson"}*) – The function to measure the quality of a split. Supported criteria are:
  - "squared_error" for the mean squared error, which is equal to variance reduction as feature selection criterion and minimizes the L2 loss using the mean of each terminal node
  - "friedman_mse", which uses mean squared error with Friedman"s improvement score for potential splits
  - "absolute_error" for the mean absolute error, which minimizes the L1 loss using the median of each terminal node,
  - "poisson" which uses reduction in Poisson deviance to find splits.
- **max_features** (*int, float or {"auto", "sqrt", "log2"}*) – The number of features to consider when looking for the best split:
  - If int, then consider max_features features at each split.
  - If float, then max_features is a fraction and int(max_features * n_features) features are considered at each split.
  - If "auto", then max_features=sqrt(n_features).
  - If "sqrt", then max_features=sqrt(n_features).
  - If "log2", then max_features=log2(n_features).
  - If None, then max_features = n_features.

  The search for a split does not stop until at least one valid partition of the node samples is found, even if it requires to effectively inspect more than max_features features.
- **max_depth** (*int*) – The maximum depth of the tree. Defaults to 6.
- **min_samples_split** (*int or float*) – The minimum number of samples required to split an internal node:
  - If int, then consider min_samples_split as the minimum number.
  - If float, then min_samples_split is a fraction and ceil(min_samples_split * n_samples) are the minimum number of samples for each split.

  Defaults to 2.
- **min_weight_fraction_leaf** (*float*) – The minimum weighted fraction of the sum total of weights (of all the input samples) required to be at a leaf node. Defaults to 0.0.
- **random_seed** (*int*) – Seed for the random number generator. Defaults to 0.

**Attributes**

| | |
|---|---|
| **hyper-parameter_ranges** | { "criterion": ["squared_error", "friedman_mse", "absolute_error"], "max_features": ["auto", "sqrt", "log2"], "max_depth": Integer(4, 10),} |
| **model_family** | ModelFamily.DECISION_TREE |
| **modifies_features** | True |
| **modifies_target** | False |
| **name** | Decision Tree Regressor |
| **supported_problem_types** | [ ProblemTypes.REGRESSION, ProblemTypes.TIME_SERIES_REGRESSION,] |
| **training_only** | False |

**Methods**

| | |
|---|---|
| *clone* | Constructs a new component with the same parameters and random state. |
| *default_parameters* | Returns the default parameters for this component. |
| *describe* | Describe a component and its parameters. |
| *feature_importance* | Returns importance associated with each feature. |
| *fit* | Fits estimator to data. |
| *get_prediction_intervals* | Find the prediction intervals using the fitted regressor. |
| *load* | Loads component at file path. |
| *needs_fitting* | Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances. |
| *parameters* | Returns the parameters which were used to initialize the component. |
| *predict* | Make predictions using selected features. |
| *predict_proba* | Make probability estimates for labels. |
| *save* | Saves component at file path. |
| *update_parameters* | Updates the parameter dictionary of the component. |

**clone**(*self*)

Constructs a new component with the same parameters and random state.

**Returns** A new instance of this component with identical parameters and random state.

**default_parameters**(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

**Returns** Default parameters for this component.

**Return type** dict

**describe**(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

**Parameters**

- **print_name** (*bool, optional*) – whether to print name of component

- **return_dict** (*bool, optional*) – whether to return description as dictionary in the format {"name": name, "parameters": parameters}

**Returns** Returns dictionary if return_dict is True, else None.

**Return type** None or dict

**property feature_importance**(*self*) → pandas.Series

Returns importance associated with each feature.

**Returns** Importance associated with each feature.

**Return type** np.ndarray

**Raises** **MethodPropertyNotFoundError** – If estimator does not have a feature_importance method or a component_obj that implements feature_importance.

**fit**(*self, X: pandas.DataFrame, y: Optional[pandas.Series] = None*)

Fits estimator to data.

**Parameters**

- **X** (*pd.DataFrame*) – The input training data of shape [n_samples, n_features].
- **y** (*pd.Series, optional*) – The target training data of length [n_samples].

**Returns** self

**get_prediction_intervals**(*self, X: pandas.DataFrame, y: Optional[pandas.Series] = None, coverage: List[float] = None, predictions: pandas.Series = None*) → Dict[str, pandas.Series]

Find the prediction intervals using the fitted regressor.

This function takes the predictions of the fitted estimator and calculates the rolling standard deviation across all predictions using a window size of 5. The lower and upper predictions are determined by taking the percent point (quantile) function of the lower tail probability at each bound multiplied by the rolling standard deviation.

**Parameters**

- **X** (*pd.DataFrame*) – Data of shape [n_samples, n_features].
- **y** (*pd.Series*) – Target data. Ignored.
- **coverage** (*list[float]*) – A list of floats between the values 0 and 1 that the upper and lower bounds of the prediction interval should be calculated for.
- **predictions** (*pd.Series*) – Optional list of predictions to use. If None, will generate predictions using *X*.

**Returns** Prediction intervals, keys are in the format {coverage}_lower or {coverage}_upper.

**Return type** dict

**Raises** **MethodPropertyNotFoundError** – If the estimator does not support Time Series Regression as a problem type.

**static load**(*file_path*)

Loads component at file path.

**Parameters** **file_path** (*str*) – Location to load file.

**Returns** ComponentBase object

**needs_fitting**(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need to be fit or whose fit methods do nothing.

**Returns** True.

**property parameters**(*self*)

Returns the parameters which were used to initialize the component.

**predict**(*self*, *X: pandas.DataFrame*) → pandas.Series

Make predictions using selected features.

**Parameters** **X** (*pd.DataFrame*) – Data of shape [n_samples, n_features].

**Returns** Predicted values.

**Return type** pd.Series

**Raises** **MethodPropertyNotFoundError** – If estimator does not have a predict method or a component_obj that implements predict.

**predict_proba**(*self*, *X: pandas.DataFrame*) → pandas.Series

Make probability estimates for labels.

**Parameters** **X** (*pd.DataFrame*) – Features.

**Returns** Probability estimates.

**Return type** pd.Series

**Raises** **MethodPropertyNotFoundError** – If estimator does not have a predict_proba method or a component_obj that implements predict_proba.

**save**(*self*, *file_path*, *pickle_protocol=cloudpickle.DEFAULT_PROTOCOL*)

Saves component at file path.

**Parameters**

- **file_path** (*str*) – Location to save file.
- **pickle_protocol** (*int*) – The pickle data stream format.

**update_parameters**(*self*, *update_dict*, *reset_fit=True*)

Updates the parameter dictionary of the component.

**Parameters**

- **update_dict** (*dict*) – A dict of parameters to update.
- **reset_fit** (*bool, optional*) – If True, will set *_is_fitted* to False.

**class** evalml.pipelines.components.**DFSTransformer**(*index='index'*, *features=None*, *random_seed=0*, ***kwargs*)

Featuretools DFS component that generates features for the input features.

**Parameters**

- **index** (*string*) – The name of the column that contains the indices. If no column with this name exists, then featuretools.EntitySet() creates a column with this name to serve as the index column. Defaults to 'index'.
- **random_seed** (*int*) – Seed for the random number generator. Defaults to 0.

- **features** (*list*) – List of features to run DFS on. Defaults to None. Features will only be computed if the columns used by the feature exist in the input and if the feature itself is not in input. If features is an empty list, no transformation will occur to inputted data.

**Attributes**

| | |
|---|---|
| **hyper-parameter_ranges** | {} |
| **modifies_features** | True |
| **modifies_target** | False |
| **name** | DFS Transformer |
| **training_only** | False |

**Methods**

| | |
|---|---|
| *clone* | Constructs a new component with the same parameters and random state. |
| *contains_pre_existing_features* | Determines whether or not features from a DFS Transformer match pipeline input features. |
| *default_parameters* | Returns the default parameters for this component. |
| *describe* | Describe a component and its parameters. |
| *fit* | Fits the DFSTransformer Transformer component. |
| *fit_transform* | Fits on X and transforms X. |
| *load* | Loads component at file path. |
| *needs_fitting* | Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances. |
| *parameters* | Returns the parameters which were used to initialize the component. |
| *save* | Saves component at file path. |
| *transform* | Computes the feature matrix for the input X using featuretools' dfs algorithm. |
| *update_parameters* | Updates the parameter dictionary of the component. |

**clone**(*self*)

Constructs a new component with the same parameters and random state.

**Returns** A new instance of this component with identical parameters and random state.

**static contains_pre_existing_features**(*dfs_features: Optional[List[featuretools.feature_base.FeatureBase]], input_feature_names: List[str], target: Optional[str] = None*)

Determines whether or not features from a DFS Transformer match pipeline input features.

**Parameters**

- **dfs_features** (*Optional[List[FeatureBase]]*) – List of features output from a DFS Transformer.
- **input_feature_names** (*List[str]*) – List of input features into the DFS Transformer.

- **target** (*Optional[str]*) – The target whose values we are trying to predict. This is used to know which column to ignore if the target column is present in the list of features in the DFS Transformer's parameters.

**default_parameters**(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

**Returns** Default parameters for this component.

**Return type** dict

**describe**(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

**Parameters**

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description as dictionary in the format {"name": name, "parameters": parameters}

**Returns** Returns dictionary if return_dict is True, else None.

**Return type** None or dict

**fit**(*self*, *X*, *y=None*)

Fits the DFSTransformer Transformer component.

**Parameters**

- **X** (*pd.DataFrame, np.array*) – The input data to transform, of shape [n_samples, n_features].
- **y** (*pd.Series*) – The target training data of length [n_samples].

**Returns** self

**fit_transform**(*self*, *X*, *y=None*)

Fits on X and transforms X.

**Parameters**

- **X** (*pd.DataFrame*) – Data to fit and transform.
- **y** (*pd.Series*) – Target data.

**Returns** Transformed X.

**Return type** pd.DataFrame

**Raises** **MethodPropertyNotFoundError** – If transformer does not have a transform method or a component_obj that implements transform.

**static load**(*file_path*)

Loads component at file path.

**Parameters** **file_path** (*str*) – Location to load file.

**Returns** ComponentBase object

**needs_fitting**(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need to be fit or whose fit methods do nothing.

**Returns** True.

**property parameters**(*self*)

Returns the parameters which were used to initialize the component.

**save**(*self*, *file_path*, *pickle_protocol=cloudpickle.DEFAULT_PROTOCOL*)

Saves component at file path.

**Parameters**

- **file_path** (*str*) – Location to save file.
- **pickle_protocol** (*int*) – The pickle data stream format.

**transform**(*self*, *X*, *y=None*)

Computes the feature matrix for the input X using featuretools' dfs algorithm.

**Parameters**

- **X** (*pd.DataFrame or np.ndarray*) – The input training data to transform. Has shape [n_samples, n_features]
- **y** (*pd.Series, optional*) – Ignored.

**Returns** Feature matrix

**Return type** pd.DataFrame

**update_parameters**(*self*, *update_dict*, *reset_fit=True*)

Updates the parameter dictionary of the component.

**Parameters**

- **update_dict** (*dict*) – A dict of parameters to update.
- **reset_fit** (*bool, optional*) – If True, will set *_is_fitted* to False.

**class** evalml.pipelines.components.**DropColumns**(*columns=None*, *random_seed=0*, ***kwargs*)

Drops specified columns in input data.

**Parameters**

- **columns** (*list(string)*) – List of column names, used to determine which columns to drop.
- **random_seed** (*int*) – Seed for the random number generator. Defaults to 0.

**Attributes**

| | |
|---|---|
| **hyper-parameter_ranges** | {} |
| **modifies_features** | True |
| **modifies_target** | False |
| **name** | Drop Columns Transformer |
| **needs_fitting** | False |
| **training_only** | False |

**Methods**

| | |
|---|---|
| *clone* | Constructs a new component with the same parameters and random state. |
| *default_parameters* | Returns the default parameters for this component. |
| *describe* | Describe a component and its parameters. |
| *fit* | Fits the transformer by checking if column names are present in the dataset. |
| *fit_transform* | Fits on X and transforms X. |
| *load* | Loads component at file path. |
| *parameters* | Returns the parameters which were used to initialize the component. |
| *save* | Saves component at file path. |
| *transform* | Transforms data X by dropping columns. |
| *update_parameters* | Updates the parameter dictionary of the component. |

**clone**(*self*)

Constructs a new component with the same parameters and random state.

**Returns** A new instance of this component with identical parameters and random state.

**default_parameters**(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

**Returns** Default parameters for this component.

**Return type** dict

**describe**(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

**Parameters**

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description as dictionary in the format {"name": name, "parameters": parameters}

**Returns** Returns dictionary if return_dict is True, else None.

**Return type** None or dict

**fit**(*self*, *X*, *y=None*)

Fits the transformer by checking if column names are present in the dataset.

**Parameters**

- **X** (*pd.DataFrame*) – Data to check.
- **y** (*pd.Series, ignored*) – Targets.

**Returns** self

**fit_transform**(*self*, *X*, *y=None*)

Fits on X and transforms X.

**Parameters**

- **X** (*pd.DataFrame*) – Data to fit and transform.
- **y** (*pd.Series*) – Target data.

**Returns** Transformed X.

**Return type** pd.DataFrame

**Raises** **MethodPropertyNotFoundError** – If transformer does not have a transform method or a component_obj that implements transform.

**static load**(*file_path*)

Loads component at file path.

**Parameters** **file_path** (*str*) – Location to load file.

**Returns** ComponentBase object

**property parameters**(*self*)

Returns the parameters which were used to initialize the component.

**save**(*self*, *file_path*, *pickle_protocol=cloudpickle.DEFAULT_PROTOCOL*)

Saves component at file path.

**Parameters**

- **file_path** (*str*) – Location to save file.
- **pickle_protocol** (*int*) – The pickle data stream format.

**transform**(*self*, *X*, *y=None*)

Transforms data X by dropping columns.

**Parameters**

- **X** (*pd.DataFrame*) – Data to transform.
- **y** (*pd.Series, optional*) – Targets.

**Returns** Transformed X.

**Return type** pd.DataFrame

**update_parameters**(*self*, *update_dict*, *reset_fit=True*)

Updates the parameter dictionary of the component.

**Parameters**

- **update_dict** (*dict*) – A dict of parameters to update.
- **reset_fit** (*bool, optional*) – If True, will set *_is_fitted* to False.

**class** evalml.pipelines.components.**DropNaNRowsTransformer**(*parameters=None*, *component_obj=None*, *random_seed=0*, ***kwargs*)

Transformer to drop rows with NaN values.

**Parameters** **random_seed** (*int*) – Seed for the random number generator. Is not used by this component. Defaults to 0.

**Attributes**

| | |
|---|---|
| **hyper-parameter_ranges** | {} |
| **modifies_features** | True |
| **modifies_target** | True |
| **name** | Drop NaN Rows Transformer |
| **training_only** | False |

**Methods**

| | |
|---|---|
| *clone* | Constructs a new component with the same parameters and random state. |
| *default_parameters* | Returns the default parameters for this component. |
| *describe* | Describe a component and its parameters. |
| *fit* | Fits component to data. |
| *fit_transform* | Fits on X and transforms X. |
| *load* | Loads component at file path. |
| *needs_fitting* | Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances. |
| *parameters* | Returns the parameters which were used to initialize the component. |
| *save* | Saves component at file path. |
| *transform* | Transforms data using fitted component. |
| *update_parameters* | Updates the parameter dictionary of the component. |

**clone**(*self*)

Constructs a new component with the same parameters and random state.

**Returns** A new instance of this component with identical parameters and random state.

**default_parameters**(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

**Returns** Default parameters for this component.

**Return type** dict

**describe**(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

**Parameters**

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description as dictionary in the format {"name": name, "parameters": parameters}

**Returns** Returns dictionary if return_dict is True, else None.

**Return type** None or dict

**fit**(*self*, *X*, *y=None*)

Fits component to data.

**Parameters**

- **X** (*pd.DataFrame*) – The input training data of shape [n_samples, n_features].
- **y** (*pd.Series, optional*) – The target training data of length [n_samples].

**Returns** self

**fit_transform**(*self*, *X*, *y=None*)

Fits on X and transforms X.

**Parameters**

- **X** (*pd.DataFrame*) – Data to fit and transform.
- **y** (*pd.Series*) – Target data.

**Returns** Transformed X.

**Return type** pd.DataFrame

**Raises** **MethodPropertyNotFoundError** – If transformer does not have a transform method or a component_obj that implements transform.

**static load**(*file_path*)

Loads component at file path.

**Parameters** **file_path** (*str*) – Location to load file.

**Returns** ComponentBase object

**needs_fitting**(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need to be fit or whose fit methods do nothing.

**Returns** True.

**property parameters**(*self*)

Returns the parameters which were used to initialize the component.

**save**(*self*, *file_path*, *pickle_protocol=cloudpickle.DEFAULT_PROTOCOL*)

Saves component at file path.

**Parameters**

- **file_path** (*str*) – Location to save file.
- **pickle_protocol** (*int*) – The pickle data stream format.

**transform**(*self*, *X*, *y=None*)

Transforms data using fitted component.

**Parameters**

- **X** (*pd.DataFrame*) – Features.
- **y** (*pd.Series, optional*) – Target data.

**Returns** Data with NaN rows dropped.

**Return type** (pd.DataFrame, pd.Series)

**update_parameters**(*self*, *update_dict*, *reset_fit=True*)

Updates the parameter dictionary of the component.

**Parameters**

- **update_dict** (*dict*) – A dict of parameters to update.
- **reset_fit** (*bool, optional*) – If True, will set *_is_fitted* to False.

**class** evalml.pipelines.components.**DropNullColumns**(*pct_null_threshold=1.0*, *random_seed=0*, ***kwargs*)

Transformer to drop features whose percentage of NaN values exceeds a specified threshold.

**Parameters**

- **pct_null_threshold** (*float*) – The percentage of NaN values in an input feature to drop. Must be a value between [0, 1] inclusive. If equal to 0.0, will drop columns with any null values. If equal to 1.0, will drop columns with all null values. Defaults to 0.95.
- **random_seed** (*int*) – Seed for the random number generator. Defaults to 0.

**Attributes**

| | |
|---|---|
| **hyper-parameter_ranges** | {} |
| **modifies_features** | True |
| **modifies_target** | False |
| **name** | Drop Null Columns Transformer |
| **training_only** | False |

**Methods**

| | |
|---|---|
| *clone* | Constructs a new component with the same parameters and random state. |
| *default_parameters* | Returns the default parameters for this component. |
| *describe* | Describe a component and its parameters. |
| *fit* | Fits component to data. |
| *fit_transform* | Fits on X and transforms X. |
| *load* | Loads component at file path. |
| *needs_fitting* | Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances. |
| *parameters* | Returns the parameters which were used to initialize the component. |
| *save* | Saves component at file path. |
| *transform* | Transforms data X by dropping columns that exceed the threshold of null values. |
| *update_parameters* | Updates the parameter dictionary of the component. |

**clone**(*self*)

Constructs a new component with the same parameters and random state.

**Returns** A new instance of this component with identical parameters and random state.

**default_parameters**(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

**Returns** Default parameters for this component.

**Return type** dict

**describe**(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

**Parameters**

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description as dictionary in the format {"name": name, "parameters": parameters}

**Returns** Returns dictionary if return_dict is True, else None.

**Return type** None or dict

**fit**(*self*, *X*, *y=None*)

Fits component to data.

**Parameters**

- **X** (*pd.DataFrame*) – The input training data of shape [n_samples, n_features].
- **y** (*pd.Series, optional*) – The target training data of length [n_samples].

**Returns** self

**fit_transform**(*self*, *X*, *y=None*)

Fits on X and transforms X.

**Parameters**

- **X** (*pd.DataFrame*) – Data to fit and transform.
- **y** (*pd.Series*) – Target data.

**Returns** Transformed X.

**Return type** pd.DataFrame

**Raises** **MethodPropertyNotFoundError** – If transformer does not have a transform method or a component_obj that implements transform.

**static load**(*file_path*)

Loads component at file path.

**Parameters** **file_path** (*str*) – Location to load file.

**Returns** ComponentBase object

**needs_fitting**(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need to be fit or whose fit methods do nothing.

**Returns** True.

**property parameters**(*self*)

Returns the parameters which were used to initialize the component.

**save**(*self*, *file_path*, *pickle_protocol=cloudpickle.DEFAULT_PROTOCOL*)

Saves component at file path.

**Parameters**

- **file_path** (*str*) – Location to save file.
- **pickle_protocol** (*int*) – The pickle data stream format.

**transform**(*self*, *X*, *y=None*)

Transforms data X by dropping columns that exceed the threshold of null values.

**Parameters**

- **X** (*pd.DataFrame*) – Data to transform
- **y** (*pd.Series, optional*) – Ignored.

**Returns** Transformed X

**Return type** pd.DataFrame

**update_parameters**(*self*, *update_dict*, *reset_fit=True*)

Updates the parameter dictionary of the component.

**Parameters**

- **update_dict** (*dict*) – A dict of parameters to update.
- **reset_fit** (*bool, optional*) – If True, will set *_is_fitted* to False.

**class** evalml.pipelines.components.**DropRowsTransformer**(*indices_to_drop=None*, *random_seed=0*)

Transformer to drop rows specified by row indices.

**Parameters**

- **indices_to_drop** (*list*) – List of indices to drop in the input data. Defaults to None.

- **random_seed** (*int*) – Seed for the random number generator. Is not used by this component. Defaults to 0.

**Attributes**

| | |
|---|---|
| **hyper-parameter_ranges** | {} |
| **modifies_features** | True |
| **modifies_target** | True |
| **name** | Drop Rows Transformer |
| **training_only** | True |

**Methods**

| | |
|---|---|
| *clone* | Constructs a new component with the same parameters and random state. |
| *default_parameters* | Returns the default parameters for this component. |
| *describe* | Describe a component and its parameters. |
| *fit* | Fits component to data. |
| *fit_transform* | Fits on X and transforms X. |
| *load* | Loads component at file path. |
| *needs_fitting* | Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances. |
| *parameters* | Returns the parameters which were used to initialize the component. |
| *save* | Saves component at file path. |
| *transform* | Transforms data using fitted component. |
| *update_parameters* | Updates the parameter dictionary of the component. |

**clone**(*self*)

Constructs a new component with the same parameters and random state.

**Returns** A new instance of this component with identical parameters and random state.

**default_parameters**(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

**Returns** Default parameters for this component.

**Return type** dict

**describe**(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

**Parameters**

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description as dictionary in the format {"name": name, "parameters": parameters}

**Returns** Returns dictionary if return_dict is True, else None.

**Return type** None or dict

`fit`(*self*, *X*, *y=None*)

Fits component to data.

**Parameters**

- **X** (*pd.DataFrame*) – The input training data of shape [n_samples, n_features].
- **y** (*pd.Series, optional*) – The target training data of length [n_samples].

**Returns** self

**Raises** **ValueError** – If indices to drop do not exist in input features or target.

`fit_transform`(*self*, *X*, *y=None*)

Fits on X and transforms X.

**Parameters**

- **X** (*pd.DataFrame*) – Data to fit and transform.
- **y** (*pd.Series*) – Target data.

**Returns** Transformed X.

**Return type** pd.DataFrame

**Raises** **MethodPropertyNotFoundError** – If transformer does not have a transform method or a component_obj that implements transform.

`static load`(*file_path*)

Loads component at file path.

**Parameters** **file_path** (*str*) – Location to load file.

**Returns** ComponentBase object

`needs_fitting`(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need to be fit or whose fit methods do nothing.

**Returns** True.

`property parameters`(*self*)

Returns the parameters which were used to initialize the component.

`save`(*self*, *file_path*, *pickle_protocol=cloudpickle.DEFAULT_PROTOCOL*)

Saves component at file path.

**Parameters**

- **file_path** (*str*) – Location to save file.
- **pickle_protocol** (*int*) – The pickle data stream format.

`transform`(*self*, *X*, *y=None*)

Transforms data using fitted component.

**Parameters**

- **X** (*pd.DataFrame*) – Features.

- **y** (*pd.Series, optional*) – Target data.

**Returns** Data with row indices dropped.

**Return type** (pd.DataFrame, pd.Series)

**update_parameters**(*self*, *update_dict*, *reset_fit=True*)

Updates the parameter dictionary of the component.

**Parameters**

- **update_dict** (*dict*) – A dict of parameters to update.
- **reset_fit** (*bool, optional*) – If True, will set *_is_fitted* to False.

**class** evalml.pipelines.components.**ElasticNetClassifier**(*penalty='elasticnet'*, *C=1.0*, *l1_ratio=0.15*, *multi_class='auto'*, *solver='saga'*, *n_jobs=-1*, *random_seed=0*, ***kwargs*)

Elastic Net Classifier. Uses Logistic Regression with elasticnet penalty as the base estimator.

**Parameters**

- **penalty** (*{"l1", "l2", "elasticnet", "none"}*) – The norm used in penalization. Defaults to "elasticnet".
- **C** (*float*) – Inverse of regularization strength. Must be a positive float. Defaults to 1.0.
- **l1_ratio** (*float*) – The mixing parameter, with 0 <= l1_ratio <= 1. Only used if penalty='elasticnet'. Setting l1_ratio=0 is equivalent to using penalty='l2', while setting l1_ratio=1 is equivalent to using penalty='l1'. For 0 < l1_ratio <1, the penalty is a combination of L1 and L2. Defaults to 0.15.
- **multi_class** (*{"auto", "ovr", "multinomial"}*) – If the option chosen is "ovr", then a binary problem is fit for each label. For "multinomial" the loss minimised is the multinomial loss fit across the entire probability distribution, even when the data is binary. "multinomial" is unavailable when solver="liblinear". "auto" selects "ovr" if the data is binary, or if solver="liblinear", and otherwise selects "multinomial". Defaults to "auto".
- **solver** (*{"newton-cg", "lbfgs", "liblinear", "sag", "saga"}*) – Algorithm to use in the optimization problem. For small datasets, "liblinear" is a good choice, whereas "sag" and "saga" are faster for large ones. For multiclass problems, only "newton-cg", "sag", "saga" and "lbfgs" handle multinomial loss; "liblinear" is limited to one-versus-rest schemes.
  - "newton-cg", "lbfgs", "sag" and "saga" handle L2 or no penalty
  - "liblinear" and "saga" also handle L1 penalty
  - "saga" also supports "elasticnet" penalty
  - "liblinear" does not support setting penalty='none'

  Defaults to "saga".
- **n_jobs** (*int*) – Number of parallel threads used to run xgboost. Note that creating thread contention will significantly slow down the algorithm. Defaults to -1.
- **random_seed** (*int*) – Seed for the random number generator. Defaults to 0.

**Attributes**

| | |
|---|---|
| **hyper-parameter_ranges** | { "C": Real(0.01, 10), "l1_ratio": Real(0, 1)} |
| **model_family** | ModelFamily.LINEAR_MODEL |
| **modifies_features** | True |
| **modifies_target** | False |
| **name** | Elastic Net Classifier |
| **supported_problem_types** | [ ProblemTypes.BINARY, ProblemTypes.MULTICLASS, ProblemTypes.TIME_SERIES_BINARY, ProblemTypes.TIME_SERIES_MULTICLASS,] |
| **training_only** | False |

**Methods**

| | |
|---|---|
| *clone* | Constructs a new component with the same parameters and random state. |
| *default_parameters* | Returns the default parameters for this component. |
| *describe* | Describe a component and its parameters. |
| *feature_importance* | Feature importance for fitted ElasticNet classifier. |
| *fit* | Fits ElasticNet classifier component to data. |
| *get_prediction_intervals* | Find the prediction intervals using the fitted regressor. |
| *load* | Loads component at file path. |
| *needs_fitting* | Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances. |
| *parameters* | Returns the parameters which were used to initialize the component. |
| *predict* | Make predictions using selected features. |
| *predict_proba* | Make probability estimates for labels. |
| *save* | Saves component at file path. |
| *update_parameters* | Updates the parameter dictionary of the component. |

**clone**(*self*)

Constructs a new component with the same parameters and random state.

**Returns** A new instance of this component with identical parameters and random state.

**default_parameters**(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

**Returns** Default parameters for this component.

**Return type** dict

**describe**(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

**Parameters**

- **print_name** (*bool, optional*) – whether to print name of component

- **return_dict** (*bool, optional*) – whether to return description as dictionary in the format {"name": name, "parameters": parameters}

**Returns** Returns dictionary if return_dict is True, else None.

**Return type** None or dict

**property feature_importance**(*self*)

Feature importance for fitted ElasticNet classifier.

**fit**(*self*, *X*, *y*)

Fits ElasticNet classifier component to data.

**Parameters**

- **X** (*pd.DataFrame*) – The input training data of shape [n_samples, n_features].
- **y** (*pd.Series*) – The target training data of length [n_samples].

**Returns** self

**get_prediction_intervals**(*self*, *X: pandas.DataFrame*, *y: Optional[pandas.Series] = None*, *coverage: List[float] = None*, *predictions: pandas.Series = None*) → Dict[str, pandas.Series]

Find the prediction intervals using the fitted regressor.

This function takes the predictions of the fitted estimator and calculates the rolling standard deviation across all predictions using a window size of 5. The lower and upper predictions are determined by taking the percent point (quantile) function of the lower tail probability at each bound multiplied by the rolling standard deviation.

**Parameters**

- **X** (*pd.DataFrame*) – Data of shape [n_samples, n_features].
- **y** (*pd.Series*) – Target data. Ignored.
- **coverage** (*list[float]*) – A list of floats between the values 0 and 1 that the upper and lower bounds of the prediction interval should be calculated for.
- **predictions** (*pd.Series*) – Optional list of predictions to use. If None, will generate predictions using *X*.

**Returns** Prediction intervals, keys are in the format {coverage}_lower or {coverage}_upper.

**Return type** dict

**Raises** **MethodPropertyNotFoundError** – If the estimator does not support Time Series Regression as a problem type.

**static load**(*file_path*)

Loads component at file path.

**Parameters** **file_path** (*str*) – Location to load file.

**Returns** ComponentBase object

**needs_fitting**(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need to be fit or whose fit methods do nothing.

**Returns** True.

**property parameters**(*self*)

Returns the parameters which were used to initialize the component.

**predict**(*self*, *X: pandas.DataFrame*) → pandas.Series

Make predictions using selected features.

**Parameters** **X** (*pd.DataFrame*) – Data of shape [n_samples, n_features].

**Returns** Predicted values.

**Return type** pd.Series

**Raises** **MethodPropertyNotFoundError** – If estimator does not have a predict method or a component_obj that implements predict.

**predict_proba**(*self*, *X: pandas.DataFrame*) → pandas.Series

Make probability estimates for labels.

**Parameters** **X** (*pd.DataFrame*) – Features.

**Returns** Probability estimates.

**Return type** pd.Series

**Raises** **MethodPropertyNotFoundError** – If estimator does not have a predict_proba method or a component_obj that implements predict_proba.

**save**(*self*, *file_path*, *pickle_protocol=cloudpickle.DEFAULT_PROTOCOL*)

Saves component at file path.

**Parameters**

- **file_path** (*str*) – Location to save file.
- **pickle_protocol** (*int*) – The pickle data stream format.

**update_parameters**(*self*, *update_dict*, *reset_fit=True*)

Updates the parameter dictionary of the component.

**Parameters**

- **update_dict** (*dict*) – A dict of parameters to update.
- **reset_fit** (*bool, optional*) – If True, will set *_is_fitted* to False.

**class** evalml.pipelines.components.**ElasticNetRegressor**(*alpha=0.0001*, *l1_ratio=0.15*, *max_iter=1000*, *random_seed=0*, ***kwargs*)

Elastic Net Regressor.

**Parameters**

- **alpha** (*float*) – Constant that multiplies the penalty terms. Defaults to 0.0001.
- **l1_ratio** (*float*) – The mixing parameter, with 0 <= l1_ratio <= 1. Only used if penalty='elasticnet'. Setting l1_ratio=0 is equivalent to using penalty='l2', while setting l1_ratio=1 is equivalent to using penalty='l1'. For 0 < l1_ratio <1, the penalty is a combination of L1 and L2. Defaults to 0.15.
- **max_iter** (*int*) – The maximum number of iterations. Defaults to 1000.
- **random_seed** (*int*) – Seed for the random number generator. Defaults to 0.

**Attributes**

| | |
|---|---|
| **hyper-parameter_ranges** | { "alpha": Real(0, 1), "l1_ratio": Real(0, 1),} |
| **model_family** | ModelFamily.LINEAR_MODEL |
| **modifies_features** | True |
| **modifies_target** | False |
| **name** | Elastic Net Regressor |
| **supported_problem_types** | [ ProblemTypes.REGRESSION, ProblemTypes.TIME_SERIES_REGRESSION,] |
| **training_only** | False |

**Methods**

| | |
|---|---|
| *clone* | Constructs a new component with the same parameters and random state. |
| *default_parameters* | Returns the default parameters for this component. |
| *describe* | Describe a component and its parameters. |
| *feature_importance* | Feature importance for fitted ElasticNet regressor. |
| *fit* | Fits estimator to data. |
| *get_prediction_intervals* | Find the prediction intervals using the fitted regressor. |
| *load* | Loads component at file path. |
| *needs_fitting* | Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances. |
| *parameters* | Returns the parameters which were used to initialize the component. |
| *predict* | Make predictions using selected features. |
| *predict_proba* | Make probability estimates for labels. |
| *save* | Saves component at file path. |
| *update_parameters* | Updates the parameter dictionary of the component. |

`clone`(*self*)

Constructs a new component with the same parameters and random state.

**Returns** A new instance of this component with identical parameters and random state.

`default_parameters`(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

**Returns** Default parameters for this component.

**Return type** dict

`describe`(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

**Parameters**

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description as dictionary in the format {"name": name, "parameters": parameters}

**Returns** Returns dictionary if return_dict is True, else None.

**Return type** None or dict

**property feature_importance**(*self*)

Feature importance for fitted ElasticNet regressor.

**fit**(*self*, *X: pandas.DataFrame*, *y: Optional[pandas.Series] = None*)

Fits estimator to data.

**Parameters**

- **X** (*pd.DataFrame*) – The input training data of shape [n_samples, n_features].
- **y** (*pd.Series, optional*) – The target training data of length [n_samples].

**Returns** self

**get_prediction_intervals**(*self*, *X: pandas.DataFrame*, *y: Optional[pandas.Series] = None*, *coverage: List[float] = None*, *predictions: pandas.Series = None*) → Dict[str, pandas.Series]

Find the prediction intervals using the fitted regressor.

This function takes the predictions of the fitted estimator and calculates the rolling standard deviation across all predictions using a window size of 5. The lower and upper predictions are determined by taking the percent point (quantile) function of the lower tail probability at each bound multiplied by the rolling standard deviation.

**Parameters**

- **X** (*pd.DataFrame*) – Data of shape [n_samples, n_features].
- **y** (*pd.Series*) – Target data. Ignored.
- **coverage** (*list[float]*) – A list of floats between the values 0 and 1 that the upper and lower bounds of the prediction interval should be calculated for.
- **predictions** (*pd.Series*) – Optional list of predictions to use. If None, will generate predictions using *X*.

**Returns** Prediction intervals, keys are in the format {coverage}_lower or {coverage}_upper.

**Return type** dict

**Raises** **MethodPropertyNotFoundError** – If the estimator does not support Time Series Regression as a problem type.

**static load**(*file_path*)

Loads component at file path.

**Parameters** **file_path** (*str*) – Location to load file.

**Returns** ComponentBase object

**needs_fitting**(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need to be fit or whose fit methods do nothing.

**Returns** True.

**property parameters**(*self*)

Returns the parameters which were used to initialize the component.

**predict**(*self*, *X: pandas.DataFrame*) → pandas.Series

Make predictions using selected features.

**Parameters** **X** (*pd.DataFrame*) – Data of shape [n_samples, n_features].

**Returns** Predicted values.

**Return type** pd.Series

**Raises** **MethodPropertyNotFoundError** – If estimator does not have a predict method or a component_obj that implements predict.

**predict_proba**(*self*, *X: pandas.DataFrame*) → pandas.Series

Make probability estimates for labels.

**Parameters** **X** (*pd.DataFrame*) – Features.

**Returns** Probability estimates.

**Return type** pd.Series

**Raises** **MethodPropertyNotFoundError** – If estimator does not have a predict_proba method or a component_obj that implements predict_proba.

**save**(*self*, *file_path*, *pickle_protocol=cloudpickle.DEFAULT_PROTOCOL*)

Saves component at file path.

**Parameters**

- **file_path** (*str*) – Location to save file.
- **pickle_protocol** (*int*) – The pickle data stream format.

**update_parameters**(*self*, *update_dict*, *reset_fit=True*)

Updates the parameter dictionary of the component.

**Parameters**

- **update_dict** (*dict*) – A dict of parameters to update.
- **reset_fit** (*bool, optional*) – If True, will set *_is_fitted* to False.

**class** evalml.pipelines.components.**EmailFeaturizer**(*random_seed=0*, ***kwargs*)

Transformer that can automatically extract features from emails.

**Parameters** **random_seed** (*int*) – Seed for the random number generator. Defaults to 0.

**Attributes**

| | |
|---|---|
| **hyper-parameter_ranges** | {} |
| **modifies_features** | True |
| **modifies_target** | False |
| **name** | Email Featurizer |
| **training_only** | False |

**Methods**

| | |
|---|---|
| *clone* | Constructs a new component with the same parameters and random state. |
| *default_parameters* | Returns the default parameters for this component. |
| *describe* | Describe a component and its parameters. |
| *fit* | Fits component to data. |
| *fit_transform* | Fits on X and transforms X. |
| *load* | Loads component at file path. |
| *needs_fitting* | Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances. |
| *parameters* | Returns the parameters which were used to initialize the component. |
| *save* | Saves component at file path. |
| *transform* | Transforms data X. |
| *update_parameters* | Updates the parameter dictionary of the component. |

**clone**(*self*)

Constructs a new component with the same parameters and random state.

**Returns** A new instance of this component with identical parameters and random state.

**default_parameters**(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

**Returns** Default parameters for this component.

**Return type** dict

**describe**(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

**Parameters**

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description as dictionary in the format {"name": name, "parameters": parameters}

**Returns** Returns dictionary if return_dict is True, else None.

**Return type** None or dict

**fit**(*self*, *X*, *y=None*)

Fits component to data.

**Parameters**

- **X** (*pd.DataFrame*) – The input training data of shape [n_samples, n_features]
- **y** (*pd.Series, optional*) – The target training data of length [n_samples]

**Returns** self

**Raises** **MethodPropertyNotFoundError** – If component does not have a fit method or a component_obj that implements fit.

**fit_transform**(*self*, *X*, *y=None*)

Fits on X and transforms X.

**Parameters**

- **X** (*pd.DataFrame*) – Data to fit and transform.
- **y** (*pd.Series*) – Target data.

**Returns** Transformed X.

**Return type** pd.DataFrame

**Raises** **MethodPropertyNotFoundError** – If transformer does not have a transform method or a component_obj that implements transform.

**static load**(*file_path*)

Loads component at file path.

**Parameters** **file_path** (*str*) – Location to load file.

**Returns** ComponentBase object

**needs_fitting**(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need to be fit or whose fit methods do nothing.

**Returns** True.

**property parameters**(*self*)

Returns the parameters which were used to initialize the component.

**save**(*self*, *file_path*, *pickle_protocol=cloudpickle.DEFAULT_PROTOCOL*)

Saves component at file path.

**Parameters**

- **file_path** (*str*) – Location to save file.
- **pickle_protocol** (*int*) – The pickle data stream format.

**transform**(*self*, *X*, *y=None*)

Transforms data X.

**Parameters**

- **X** (*pd.DataFrame*) – Data to transform.
- **y** (*pd.Series, optional*) – Target data.

**Returns** Transformed X

**Return type** pd.DataFrame

**Raises** **MethodPropertyNotFoundError** – If transformer does not have a transform method or a component_obj that implements transform.

**update_parameters**(*self*, *update_dict*, *reset_fit=True*)

Updates the parameter dictionary of the component.

**Parameters**

- **update_dict** (*dict*) – A dict of parameters to update.

- **reset_fit** (*bool, optional*) – If True, will set *_is_fitted* to False.

**class** evalml.pipelines.components.**Estimator**(*parameters: dict = None, component_obj: Type[evalml.pipelines.components.ComponentBase] = None, random_seed: Union[int, float] = 0, **kwargs*)

A component that fits and predicts given data.

To implement a new Estimator, define your own class which is a subclass of Estimator, including a name and a list of acceptable ranges for any parameters to be tuned during the automl search (hyperparameters). Define an *__init__* method which sets up any necessary state and objects. Make sure your *__init__* only uses standard keyword arguments and calls *super().__init__()* with a parameters dict. You may also override the *fit*, *transform*, *fit_transform* and other methods in this class if appropriate.

To see some examples, check out the definitions of any Estimator component subclass.

**Parameters**

- **parameters** (*dict*) – Dictionary of parameters for the component. Defaults to None.
- **component_obj** (*obj*) – Third-party objects useful in component implementation. Defaults to None.
- **random_seed** (*int*) – Seed for the random number generator. Defaults to 0.

**Attributes**

| | |
|---|---|
| **model_family** | ModelFamily.NONE |
| **modifies_features** | True |
| **modifies_target** | False |
| **training_only** | False |

**Methods**

| | |
|---|---|
| *clone* | Constructs a new component with the same parameters and random state. |
| *default_parameters* | Returns the default parameters for this component. |
| *describe* | Describe a component and its parameters. |
| *feature_importance* | Returns importance associated with each feature. |
| *fit* | Fits estimator to data. |
| *get_prediction_intervals* | Find the prediction intervals using the fitted regressor. |
| *load* | Loads component at file path. |
| *model_family* | ModelFamily.NONE |
| *name* | Returns string name of this component. |
| *needs_fitting* | Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances. |
| *parameters* | Returns the parameters which were used to initialize the component. |
| *predict* | Make predictions using selected features. |
| *predict_proba* | Make probability estimates for labels. |
| *save* | Saves component at file path. |
| *supported_problem_types* | Problem types this estimator supports. |
| *update_parameters* | Updates the parameter dictionary of the component. |

**clone**(*self*)

Constructs a new component with the same parameters and random state.

**Returns** A new instance of this component with identical parameters and random state.

**default_parameters**(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

**Returns** Default parameters for this component.

**Return type** dict

**describe**(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

**Parameters**

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description as dictionary in the format {"name": name, "parameters": parameters}

**Returns** Returns dictionary if return_dict is True, else None.

**Return type** None or dict

**property feature_importance**(*self*) → pandas.Series

Returns importance associated with each feature.

**Returns** Importance associated with each feature.

**Return type** np.ndarray

**Raises** **MethodPropertyNotFoundError** – If estimator does not have a feature_importance method or a component_obj that implements feature_importance.

**fit**(*self*, *X: pandas.DataFrame*, *y: Optional[pandas.Series] = None*)

Fits estimator to data.

**Parameters**

- **X** (*pd.DataFrame*) – The input training data of shape [n_samples, n_features].
- **y** (*pd.Series, optional*) – The target training data of length [n_samples].

**Returns** self

**get_prediction_intervals**(*self*, *X: pandas.DataFrame*, *y: Optional[pandas.Series] = None*, *coverage: List[float] = None*, *predictions: pandas.Series = None*) → Dict[str, pandas.Series]

Find the prediction intervals using the fitted regressor.

This function takes the predictions of the fitted estimator and calculates the rolling standard deviation across all predictions using a window size of 5. The lower and upper predictions are determined by taking the percent point (quantile) function of the lower tail probability at each bound multiplied by the rolling standard deviation.

**Parameters**

- **X** (*pd.DataFrame*) – Data of shape [n_samples, n_features].
- **y** (*pd.Series*) – Target data. Ignored.

- **coverage** (*list[float]*) – A list of floats between the values 0 and 1 that the upper and lower bounds of the prediction interval should be calculated for.
- **predictions** (*pd.Series*) – Optional list of predictions to use. If None, will generate predictions using *X*.

**Returns** Prediction intervals, keys are in the format {coverage}_lower or {coverage}_upper.

**Return type** dict

**Raises** **MethodPropertyNotFoundError** – If the estimator does not support Time Series Regression as a problem type.

**static load**(*file_path*)

Loads component at file path.

**Parameters** **file_path** (*str*) – Location to load file.

**Returns** ComponentBase object

**property model_family**(*cls*)

Returns ModelFamily of this component.

**property name**(*cls*)

Returns string name of this component.

**needs_fitting**(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need to be fit or whose fit methods do nothing.

**Returns** True.

**property parameters**(*self*)

Returns the parameters which were used to initialize the component.

**predict**(*self*, *X: pandas.DataFrame*) → pandas.Series

Make predictions using selected features.

**Parameters** **X** (*pd.DataFrame*) – Data of shape [n_samples, n_features].

**Returns** Predicted values.

**Return type** pd.Series

**Raises** **MethodPropertyNotFoundError** – If estimator does not have a predict method or a component_obj that implements predict.

**predict_proba**(*self*, *X: pandas.DataFrame*) → pandas.Series

Make probability estimates for labels.

**Parameters** **X** (*pd.DataFrame*) – Features.

**Returns** Probability estimates.

**Return type** pd.Series

**Raises** **MethodPropertyNotFoundError** – If estimator does not have a predict_proba method or a component_obj that implements predict_proba.

**save**(*self*, *file_path*, *pickle_protocol=cloudpickle.DEFAULT_PROTOCOL*)

Saves component at file path.

**Parameters**

- **file_path** (*str*) – Location to save file.
- **pickle_protocol** (*int*) – The pickle data stream format.

**property supported_problem_types**(*cls*)

Problem types this estimator supports.

**update_parameters**(*self*, *update_dict*, *reset_fit=True*)

Updates the parameter dictionary of the component.

**Parameters**

- **update_dict** (*dict*) – A dict of parameters to update.
- **reset_fit** (*bool, optional*) – If True, will set *_is_fitted* to False.

**class** evalml.pipelines.components.**ExponentialSmoothingRegressor**(*trend: Optional[str] = None*, *damped_trend: bool = False*, *seasonal: Optional[str] = None*, *sp: int = 2*, *n_jobs: int = - 1*, *random_seed: Union[int, float] = 0*, ***kwargs*)

Holt-Winters Exponential Smoothing Forecaster.

Currently ExponentialSmoothingRegressor isn't supported via conda install. It's recommended that it be installed via PyPI.

**Parameters**

- **trend** (*str*) – Type of trend component. Defaults to None.
- **damped_trend** (*bool*) – If the trend component should be damped. Defaults to False.
- **seasonal** (*str*) – Type of seasonal component. Takes one of {"additive", None}. Can also be multiplicative if
- **0** (*none of the target data is*) –
- **None.** (*but AutoMLSearch wiill not tune for this. Defaults to*) –
- **sp** (*int*) – The number of seasonal periods to consider. Defaults to 2.
- **n_jobs** (*int or None*) – Non-negative integer describing level of parallelism used for pipelines. Defaults to -1.
- **random_seed** (*int*) – Seed for the random number generator. Defaults to 0.

**Attributes**

| | |
|---|---|
| **hyper-parameter_ranges** | { "trend": [None, "additive"], "damped_trend": [True, False], "seasonal": [None, "additive"], "sp": Integer(2, 8),} |
| **model_family** | ModelFamily.EXPONENTIAL_SMOOTHING |
| **modifies_features** | True |
| **modifies_target** | False |
| **name** | Exponential Smoothing Regressor |
| **supported_problem_types** | [ProblemTypes.TIME_SERIES_REGRESSION] |
| **training_only** | False |

**Methods**

| | |
|---|---|
| *clone* | Constructs a new component with the same parameters and random state. |
| *default_parameters* | Returns the default parameters for this component. |
| *describe* | Describe a component and its parameters. |
| *feature_importance* | Returns array of 0's with a length of 1 as feature_importance is not defined for Exponential Smoothing regressor. |
| *fit* | Fits Exponential Smoothing Regressor to data. |
| *get_prediction_intervals* | Find the prediction intervals using the fitted ExponentialSmoothingRegressor. |
| *load* | Loads component at file path. |
| *needs_fitting* | Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances. |
| *parameters* | Returns the parameters which were used to initialize the component. |
| *predict* | Make predictions using fitted Exponential Smoothing regressor. |
| *predict_proba* | Make probability estimates for labels. |
| *save* | Saves component at file path. |
| *update_parameters* | Updates the parameter dictionary of the component. |

**clone**(*self*)

Constructs a new component with the same parameters and random state.

**Returns** A new instance of this component with identical parameters and random state.

**default_parameters**(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

**Returns** Default parameters for this component.

**Return type** dict

**describe**(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

**Parameters**

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description as dictionary in the format {"name": name, "parameters": parameters}

**Returns** Returns dictionary if return_dict is True, else None.

**Return type** None or dict

**property feature_importance**(*self*) → pandas.Series

Returns array of 0's with a length of 1 as feature_importance is not defined for Exponential Smoothing regressor.

**fit**(*self, X: pandas.DataFrame, y: Optional[pandas.Series] = None*)

Fits Exponential Smoothing Regressor to data.

**Parameters**

- **X** (*pd.DataFrame*) – The input training data of shape [n_samples, n_features]. Ignored.
- **y** (*pd.Series*) – The target training data of length [n_samples].

**Returns** self

**Raises** **ValueError** – If y was not passed in.

**get_prediction_intervals**(*self, X: pandas.DataFrame, y: Optional[pandas.Series] = None, coverage: List[float] = None, predictions: pandas.Series = None*) → Dict[str, pandas.Series]

Find the prediction intervals using the fitted ExponentialSmoothingRegressor.

Calculates the prediction intervals by using a simulation of the time series following a specified state space model.

**Parameters**

- **X** (*pd.DataFrame*) – Data of shape [n_samples, n_features].
- **y** (*pd.Series*) – Target data. Optional.
- **coverage** (*List[float]*) – A list of floats between the values 0 and 1 that the upper and lower bounds of the prediction interval should be calculated for.
- **predictions** (*pd.Series*) – Not used for Exponential Smoothing regressor.

**Returns** Prediction intervals, keys are in the format {coverage}_lower or {coverage}_upper.

**Return type** dict

**static load**(*file_path*)

Loads component at file path.

**Parameters** **file_path** (*str*) – Location to load file.

**Returns** ComponentBase object

**needs_fitting**(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need to be fit or whose fit methods do nothing.

**Returns** True.

**property parameters**(*self*)

Returns the parameters which were used to initialize the component.

**predict**(*self*, *X: pandas.DataFrame*, *y: Optional[pandas.Series] = None*) → pandas.Series

Make predictions using fitted Exponential Smoothing regressor.

**Parameters**

- **X** (*pd.DataFrame*) – Data of shape [n_samples, n_features]. Ignored except to set forecast horizon.
- **y** (*pd.Series*) – Target data.

**Returns** Predicted values.

**Return type** pd.Series

**predict_proba**(*self*, *X: pandas.DataFrame*) → pandas.Series

Make probability estimates for labels.

**Parameters** **X** (*pd.DataFrame*) – Features.

**Returns** Probability estimates.

**Return type** pd.Series

**Raises** **MethodPropertyNotFoundError** – If estimator does not have a predict_proba method or a component_obj that implements predict_proba.

**save**(*self*, *file_path*, *pickle_protocol=cloudpickle.DEFAULT_PROTOCOL*)

Saves component at file path.

**Parameters**

- **file_path** (*str*) – Location to save file.
- **pickle_protocol** (*int*) – The pickle data stream format.

**update_parameters**(*self*, *update_dict*, *reset_fit=True*)

Updates the parameter dictionary of the component.

**Parameters**

- **update_dict** (*dict*) – A dict of parameters to update.
- **reset_fit** (*bool, optional*) – If True, will set *_is_fitted* to False.

**class** evalml.pipelines.components.**ExtraTreesClassifier**(*n_estimators=100*, *max_features='auto'*, *max_depth=6*, *min_samples_split=2*, *min_weight_fraction_leaf=0.0*, *n_jobs=- 1*, *random_seed=0*, ***kwargs*)

Extra Trees Classifier.

**Parameters**

- **n_estimators** (*float*) – The number of trees in the forest. Defaults to 100.
- **max_features** (*int, float or {"auto", "sqrt", "log2"}*) – The number of features to consider when looking for the best split:
  - If int, then consider max_features features at each split.
  - If float, then max_features is a fraction and int(max_features * n_features) features are considered at each split.

- If "auto", then max_features=sqrt(n_features).
- If "sqrt", then max_features=sqrt(n_features).
- If "log2", then max_features=log2(n_features).
- If None, then max_features = n_features.

The search for a split does not stop until at least one valid partition of the node samples is found, even if it requires to effectively inspect more than max_features features. Defaults to "auto".

- **max_depth** (*int*) – The maximum depth of the tree. Defaults to 6.
- **min_samples_split** (*int or float*) – The minimum number of samples required to split an internal node:
  - If int, then consider min_samples_split as the minimum number.
  - If float, then min_samples_split is a fraction and ceil(min_samples_split * n_samples) are the minimum number of samples for each split.
- **2.** (*Defaults to*) –
- **min_weight_fraction_leaf** (*float*) – The minimum weighted fraction of the sum total of weights (of all the input samples) required to be at a leaf node. Defaults to 0.0.
- **n_jobs** (*int or None*) – Number of jobs to run in parallel. -1 uses all processes. Defaults to -1.
- **random_seed** (*int*) – Seed for the random number generator. Defaults to 0.

**Attributes**

| | |
|---|---|
| **hyper-parameter_ranges** | { "n_estimators": Integer(10, 1000), "max_features": ["auto", "sqrt", "log2"], "max_depth": Integer(4, 10),} |
| **model_family** | ModelFamily.EXTRA_TREES |
| **modifies_features** | True |
| **modifies_target** | False |
| **name** | Extra Trees Classifier |
| **supported_problem_types** | [ ProblemTypes.BINARY, ProblemTypes.MULTICLASS, ProblemTypes.TIME_SERIES_BINARY, ProblemTypes.TIME_SERIES_MULTICLASS,] |
| **training_only** | False |

**Methods**

| | |
|---|---|
| *clone* | Constructs a new component with the same parameters and random state. |
| *default_parameters* | Returns the default parameters for this component. |
| *describe* | Describe a component and its parameters. |
| *feature_importance* | Returns importance associated with each feature. |
| *fit* | Fits estimator to data. |
| *get_prediction_intervals* | Find the prediction intervals using the fitted regressor. |
| *load* | Loads component at file path. |
| *needs_fitting* | Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances. |
| *parameters* | Returns the parameters which were used to initialize the component. |
| *predict* | Make predictions using selected features. |
| *predict_proba* | Make probability estimates for labels. |
| *save* | Saves component at file path. |
| *update_parameters* | Updates the parameter dictionary of the component. |

**clone**(*self*)

Constructs a new component with the same parameters and random state.

**Returns** A new instance of this component with identical parameters and random state.

**default_parameters**(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

**Returns** Default parameters for this component.

**Return type** dict

**describe**(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

**Parameters**

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description as dictionary in the format {"name": name, "parameters": parameters}

**Returns** Returns dictionary if return_dict is True, else None.

**Return type** None or dict

**property feature_importance**(*self*) → pandas.Series

Returns importance associated with each feature.

**Returns** Importance associated with each feature.

**Return type** np.ndarray

**Raises** **MethodPropertyNotFoundError** – If estimator does not have a feature_importance method or a component_obj that implements feature_importance.

**fit**(*self*, *X: pandas.DataFrame*, *y: Optional[pandas.Series] = None*)

Fits estimator to data.

**Parameters**

- **X** (*pd.DataFrame*) – The input training data of shape [n_samples, n_features].
- **y** (*pd.Series, optional*) – The target training data of length [n_samples].

**Returns** self

**get_prediction_intervals**(*self, X: pandas.DataFrame, y: Optional[pandas.Series] = None, coverage: List[float] = None, predictions: pandas.Series = None*) → Dict[str, pandas.Series]

Find the prediction intervals using the fitted regressor.

This function takes the predictions of the fitted estimator and calculates the rolling standard deviation across all predictions using a window size of 5. The lower and upper predictions are determined by taking the percent point (quantile) function of the lower tail probability at each bound multiplied by the rolling standard deviation.

**Parameters**

- **X** (*pd.DataFrame*) – Data of shape [n_samples, n_features].
- **y** (*pd.Series*) – Target data. Ignored.
- **coverage** (*list[float]*) – A list of floats between the values 0 and 1 that the upper and lower bounds of the prediction interval should be calculated for.
- **predictions** (*pd.Series*) – Optional list of predictions to use. If None, will generate predictions using *X*.

**Returns** Prediction intervals, keys are in the format {coverage}_lower or {coverage}_upper.

**Return type** dict

**Raises** **MethodPropertyNotFoundError** – If the estimator does not support Time Series Regression as a problem type.

**static load**(*file_path*)

Loads component at file path.

**Parameters** **file_path** (*str*) – Location to load file.

**Returns** ComponentBase object

**needs_fitting**(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need to be fit or whose fit methods do nothing.

**Returns** True.

**property parameters**(*self*)

Returns the parameters which were used to initialize the component.

**predict**(*self, X: pandas.DataFrame*) → pandas.Series

Make predictions using selected features.

**Parameters** **X** (*pd.DataFrame*) – Data of shape [n_samples, n_features].

**Returns** Predicted values.

**Return type** pd.Series

**Raises** `MethodPropertyNotFoundError` – If estimator does not have a predict method or a component_obj that implements predict.

`predict_proba`(*self*, *X: pandas.DataFrame*) → pandas.Series

Make probability estimates for labels.

**Parameters** `X` (*pd.DataFrame*) – Features.

**Returns** Probability estimates.

**Return type** pd.Series

**Raises** `MethodPropertyNotFoundError` – If estimator does not have a predict_proba method or a component_obj that implements predict_proba.

`save`(*self*, *file_path*, *pickle_protocol=cloudpickle.DEFAULT_PROTOCOL*)

Saves component at file path.

**Parameters**

- `file_path` (*str*) – Location to save file.
- `pickle_protocol` (*int*) – The pickle data stream format.

`update_parameters`(*self*, *update_dict*, *reset_fit=True*)

Updates the parameter dictionary of the component.

**Parameters**

- `update_dict` (*dict*) – A dict of parameters to update.
- `reset_fit` (*bool, optional*) – If True, will set *_is_fitted* to False.

`class` `evalml.pipelines.components.`**`ExtraTreesRegressor`**(*n_estimators: int = 100*, *max_features: str = 'auto'*, *max_depth: int = 6*, *min_samples_split: int = 2*, *min_weight_fraction_leaf: float = 0.0*, *n_jobs: int = - 1*, *random_seed: Union[int, float] = 0*, ***kwargs*)

Extra Trees Regressor.

**Parameters**

- `n_estimators` (*float*) – The number of trees in the forest. Defaults to 100.
- `max_features` (*int, float or {"auto", "sqrt", "log2"}*) – The number of features to consider when looking for the best split:
  - If int, then consider max_features features at each split.
  - If float, then max_features is a fraction and int(max_features * n_features) features are considered at each split.
  - If "auto", then max_features=sqrt(n_features).
  - If "sqrt", then max_features=sqrt(n_features).
  - If "log2", then max_features=log2(n_features).
  - If None, then max_features = n_features.

  The search for a split does not stop until at least one valid partition of the node samples is found, even if it requires to effectively inspect more than max_features features. Defaults to "auto".
- `max_depth` (*int*) – The maximum depth of the tree. Defaults to 6.

- **min_samples_split** (*int or float*) – The minimum number of samples required to split an internal node:
  - If int, then consider min_samples_split as the minimum number.
  - If float, then min_samples_split is a fraction and ceil(min_samples_split * n_samples) are the minimum number of samples for each split.
- **2.** (*Defaults to*) –
- **min_weight_fraction_leaf** (*float*) – The minimum weighted fraction of the sum total of weights (of all the input samples) required to be at a leaf node. Defaults to 0.0.
- **n_jobs** (*int or None*) – Number of jobs to run in parallel. -1 uses all processes. Defaults to -1.
- **random_seed** (*int*) – Seed for the random number generator. Defaults to 0.

**Attributes**

| | |
|---|---|
| **hyper-parameter_ranges** | { "n_estimators": Integer(10, 1000), "max_features": ["auto", "sqrt", "log2"], "max_depth": Integer(4, 10),} |
| **model_family** | ModelFamily.EXTRA_TREES |
| **modifies_features** | True |
| **modifies_target** | False |
| **name** | Extra Trees Regressor |
| **supported_problem_types** | [ ProblemTypes.REGRESSION, ProblemTypes.TIME_SERIES_REGRESSION,] |
| **training_only** | False |

**Methods**

| | |
|---|---|
| *clone* | Constructs a new component with the same parameters and random state. |
| *default_parameters* | Returns the default parameters for this component. |
| *describe* | Describe a component and its parameters. |
| *feature_importance* | Returns importance associated with each feature. |
| *fit* | Fits estimator to data. |
| *get_prediction_intervals* | Find the prediction intervals using the fitted ExtraTreesRegressor. |
| *load* | Loads component at file path. |
| *needs_fitting* | Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances. |
| *parameters* | Returns the parameters which were used to initialize the component. |
| *predict* | Make predictions using selected features. |
| *predict_proba* | Make probability estimates for labels. |
| *save* | Saves component at file path. |
| *update_parameters* | Updates the parameter dictionary of the component. |

**clone**(*self*)

Constructs a new component with the same parameters and random state.

**Returns** A new instance of this component with identical parameters and random state.

**default_parameters**(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

**Returns** Default parameters for this component.

**Return type** dict

**describe**(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

**Parameters**

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description as dictionary in the format {"name": name, "parameters": parameters}

**Returns** Returns dictionary if return_dict is True, else None.

**Return type** None or dict

**property feature_importance**(*self*) → pandas.Series

Returns importance associated with each feature.

**Returns** Importance associated with each feature.

**Return type** np.ndarray

**Raises** **MethodPropertyNotFoundError** – If estimator does not have a feature_importance method or a component_obj that implements feature_importance.

**fit**(*self*, *X: pandas.DataFrame*, *y: Optional[pandas.Series] = None*)

Fits estimator to data.

**Parameters**

- **X** (*pd.DataFrame*) – The input training data of shape [n_samples, n_features].
- **y** (*pd.Series, optional*) – The target training data of length [n_samples].

**Returns** self

**get_prediction_intervals**(*self*, *X: pandas.DataFrame*, *y: Optional[pandas.Series] = None*, *coverage: List[float] = None*, *predictions: pandas.Series = None*) → Dict[str, pandas.Series]

Find the prediction intervals using the fitted ExtraTreesRegressor.

**Parameters**

- **X** (*pd.DataFrame*) – Data of shape [n_samples, n_features].
- **y** (*pd.Series*) – Target data. Optional.
- **coverage** (*list[float]*) – A list of floats between the values 0 and 1 that the upper and lower bounds of the prediction interval should be calculated for.
- **predictions** (*pd.Series*) – Optional list of predictions to use. If None, will generate predictions using *X*.

**Returns** Prediction intervals, keys are in the format {coverage}_lower or {coverage}_upper.

**Return type** dict

**static load**(*file_path*)

Loads component at file path.

**Parameters** **file_path** (*str*) – Location to load file.

**Returns** ComponentBase object

**needs_fitting**(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need to be fit or whose fit methods do nothing.

**Returns** True.

**property parameters**(*self*)

Returns the parameters which were used to initialize the component.

**predict**(*self*, *X: pandas.DataFrame*) → pandas.Series

Make predictions using selected features.

**Parameters** **X** (*pd.DataFrame*) – Data of shape [n_samples, n_features].

**Returns** Predicted values.

**Return type** pd.Series

**Raises** **MethodPropertyNotFoundError** – If estimator does not have a predict method or a component_obj that implements predict.

**predict_proba**(*self*, *X: pandas.DataFrame*) → pandas.Series

Make probability estimates for labels.

**Parameters** **X** (*pd.DataFrame*) – Features.

**Returns** Probability estimates.

**Return type** pd.Series

**Raises** **MethodPropertyNotFoundError** – If estimator does not have a predict_proba method or a component_obj that implements predict_proba.

**save**(*self*, *file_path*, *pickle_protocol=cloudpickle.DEFAULT_PROTOCOL*)

Saves component at file path.

**Parameters**

- **file_path** (*str*) – Location to save file.
- **pickle_protocol** (*int*) – The pickle data stream format.

**update_parameters**(*self*, *update_dict*, *reset_fit=True*)

Updates the parameter dictionary of the component.

**Parameters**

- **update_dict** (*dict*) – A dict of parameters to update.
- **reset_fit** (*bool, optional*) – If True, will set *_is_fitted* to False.

**class** evalml.pipelines.components.**FeatureSelector**(*parameters=None*, *component_obj=None*, *random_seed=0*, ***kwargs*)

Selects top features based on importance weights.

**Parameters**

- **parameters** (*dict*) – Dictionary of parameters for the component. Defaults to None.
- **component_obj** (*obj*) – Third-party objects useful in component implementation. Defaults to None.
- **random_seed** (*int*) – Seed for the random number generator. Defaults to 0.

**Attributes**

| | |
|---|---|
| **modifies_features** | True |
| **modifies_target** | False |
| **training_only** | False |

**Methods**

| | |
|---|---|
| *clone* | Constructs a new component with the same parameters and random state. |
| *default_parameters* | Returns the default parameters for this component. |
| *describe* | Describe a component and its parameters. |
| *fit* | Fits component to data. |
| *fit_transform* | Fit and transform data using the feature selector. |
| *get_names* | Get names of selected features. |
| *load* | Loads component at file path. |
| *name* | Returns string name of this component. |
| *needs_fitting* | Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances. |
| *parameters* | Returns the parameters which were used to initialize the component. |
| *save* | Saves component at file path. |
| *transform* | Transforms input data by selecting features. If the component_obj does not have a transform method, will raise an MethodPropertyNotFoundError exception. |
| *update_parameters* | Updates the parameter dictionary of the component. |

**clone**(*self*)

Constructs a new component with the same parameters and random state.

**Returns** A new instance of this component with identical parameters and random state.

**default_parameters**(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

**Returns** Default parameters for this component.

**Return type** dict

**describe**(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

**Parameters**

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description as dictionary in the format {"name": name, "parameters": parameters}

**Returns** Returns dictionary if return_dict is True, else None.

**Return type** None or dict

**fit**(*self*, *X*, *y=None*)

Fits component to data.

**Parameters**

- **X** (*pd.DataFrame*) – The input training data of shape [n_samples, n_features]
- **y** (*pd.Series, optional*) – The target training data of length [n_samples]

**Returns** self

**Raises** **MethodPropertyNotFoundError** – If component does not have a fit method or a component_obj that implements fit.

**fit_transform**(*self*, *X*, *y=None*)

Fit and transform data using the feature selector.

**Parameters**

- **X** (*pd.DataFrame*) – The input training data of shape [n_samples, n_features].
- **y** (*pd.Series, optional*) – The target training data of length [n_samples].

**Returns** Transformed data.

**Return type** pd.DataFrame

**get_names**(*self*)

Get names of selected features.

**Returns** List of the names of features selected.

**Return type** list[str]

**static load**(*file_path*)

Loads component at file path.

**Parameters** **file_path** (*str*) – Location to load file.

**Returns** ComponentBase object

**property name**(*cls*)

Returns string name of this component.

**needs_fitting**(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need to be fit or whose fit methods do nothing.

**Returns** True.

**property parameters**(*self*)

Returns the parameters which were used to initialize the component.

**save**(*self*, *file_path*, *pickle_protocol=cloudpickle.DEFAULT_PROTOCOL*)

Saves component at file path.

**Parameters**

- **file_path** (*str*) – Location to save file.
- **pickle_protocol** (*int*) – The pickle data stream format.

**transform**(*self*, *X*, *y=None*)

Transforms input data by selecting features. If the component_obj does not have a transform method, will raise an MethodPropertyNotFoundError exception.

**Parameters**

- **X** (*pd.DataFrame*) – Data to transform.
- **y** (*pd.Series, optional*) – Target data. Ignored.

**Returns** Transformed X

**Return type** pd.DataFrame

**Raises** **MethodPropertyNotFoundError** – If feature selector does not have a transform method or a component_obj that implements transform

**update_parameters**(*self*, *update_dict*, *reset_fit=True*)

Updates the parameter dictionary of the component.

**Parameters**

- **update_dict** (*dict*) – A dict of parameters to update.
- **reset_fit** (*bool, optional*) – If True, will set *_is_fitted* to False.

**class** evalml.pipelines.components.**Imputer**(*categorical_impute_strategy='most_frequent', categorical_fill_value=None, numeric_impute_strategy='mean', numeric_fill_value=None, boolean_impute_strategy='most_frequent', boolean_fill_value=None, random_seed=0, **kwargs*)

Imputes missing data according to a specified imputation strategy.

**Parameters**

- **categorical_impute_strategy** (*string*) – Impute strategy to use for string, object, boolean, categorical dtypes. Valid values include "most_frequent" and "constant".
- **numeric_impute_strategy** (*string*) – Impute strategy to use for numeric columns. Valid values include "mean", "median", "most_frequent", and "constant".
- **boolean_impute_strategy** (*string*) – Impute strategy to use for boolean columns. Valid values include "most_frequent" and "constant".
- **categorical_fill_value** (*string*) – When categorical_impute_strategy == "constant", fill_value is used to replace missing data. The default value of None will fill with the string "missing_value".
- **numeric_fill_value** (*int, float*) – When numeric_impute_strategy == "constant", fill_value is used to replace missing data. The default value of None will fill with 0.

- **boolean_fill_value** (*bool*) – When boolean_impute_strategy == "constant", fill_value is used to replace missing data. The default value of None will fill with True.
- **random_seed** (*int*) – Seed for the random number generator. Defaults to 0.

**Attributes**

| | |
|---|---|
| **hyper-parameter_ranges** | { "categorical_impute_strategy": ["most_frequent"], "numeric_impute_strategy": ["mean", "median", "most_frequent", "knn"], "boolean_impute_strategy": ["most_frequent"]} |
| **modifies_features** | True |
| **modifies_target** | False |
| **name** | Imputer |
| **training_only** | False |

**Methods**

| | |
|---|---|
| *clone* | Constructs a new component with the same parameters and random state. |
| *default_parameters* | Returns the default parameters for this component. |
| *describe* | Describe a component and its parameters. |
| *fit* | Fits imputer to data. 'None' values are converted to np.nan before imputation and are treated as the same. |
| *fit_transform* | Fits on X and transforms X. |
| *load* | Loads component at file path. |
| *needs_fitting* | Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances. |
| *parameters* | Returns the parameters which were used to initialize the component. |
| *save* | Saves component at file path. |
| *transform* | Transforms data X by imputing missing values. |
| *update_parameters* | Updates the parameter dictionary of the component. |

**clone**(*self*)

Constructs a new component with the same parameters and random state.

**Returns** A new instance of this component with identical parameters and random state.

**default_parameters**(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

**Returns** Default parameters for this component.

**Return type** dict

**describe**(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

**Parameters**

- **print_name** (*bool, optional*) – whether to print name of component

- **return_dict** (*bool, optional*) – whether to return description as dictionary in the format {"name": name, "parameters": parameters}

**Returns** Returns dictionary if return_dict is True, else None.

**Return type** None or dict

**fit**(*self*, *X*, *y=None*)

Fits imputer to data. 'None' values are converted to np.nan before imputation and are treated as the same.

**Parameters**

- **X** (*pd.DataFrame, np.ndarray*) – The input training data of shape [n_samples, n_features]
- **y** (*pd.Series, optional*) – The target training data of length [n_samples]

**Returns** self

**fit_transform**(*self*, *X*, *y=None*)

Fits on X and transforms X.

**Parameters**

- **X** (*pd.DataFrame*) – Data to fit and transform.
- **y** (*pd.Series*) – Target data.

**Returns** Transformed X.

**Return type** pd.DataFrame

**Raises** **MethodPropertyNotFoundError** – If transformer does not have a transform method or a component_obj that implements transform.

**static load**(*file_path*)

Loads component at file path.

**Parameters** **file_path** (*str*) – Location to load file.

**Returns** ComponentBase object

**needs_fitting**(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need to be fit or whose fit methods do nothing.

**Returns** True.

**property parameters**(*self*)

Returns the parameters which were used to initialize the component.

**save**(*self*, *file_path*, *pickle_protocol=cloudpickle.DEFAULT_PROTOCOL*)

Saves component at file path.

**Parameters**

- **file_path** (*str*) – Location to save file.
- **pickle_protocol** (*int*) – The pickle data stream format.

**transform**(*self*, *X*, *y=None*)

Transforms data X by imputing missing values.

**Parameters**

- **X** (*pd.DataFrame*) – Data to transform
- **y** (*pd.Series, optional*) – Ignored.

**Returns** Transformed X

**Return type** pd.DataFrame

**update_parameters**(*self*, *update_dict*, *reset_fit=True*)

Updates the parameter dictionary of the component.

**Parameters**

- **update_dict** (*dict*) – A dict of parameters to update.
- **reset_fit** (*bool, optional*) – If True, will set *_is_fitted* to False.

**class** evalml.pipelines.components.**KNeighborsClassifier**(*n_neighbors=5*, *weights='uniform'*, *algorithm='auto'*, *leaf_size=30*, *p=2*, *random_seed=0*, ***kwargs*)

K-Nearest Neighbors Classifier.

**Parameters**

- **n_neighbors** (*int*) – Number of neighbors to use by default. Defaults to 5.
- **weights** (*{'uniform', 'distance'} or callable*) – Weight function used in prediction. Can be:
  - 'uniform' : uniform weights. All points in each neighborhood are weighted equally.
  - 'distance' : weight points by the inverse of their distance. in this case, closer neighbors of a query point will have a greater influence than neighbors which are further away.
  - [callable] : a user-defined function which accepts an array of distances, and returns an array of the same shape containing the weights.

  Defaults to "uniform".
- **algorithm** (*{'auto', 'ball_tree', 'kd_tree', 'brute'}*) – Algorithm used to compute the nearest neighbors:
  - 'ball_tree' will use BallTree
  - 'kd_tree' will use KDTree
  - 'brute' will use a brute-force search.

  'auto' will attempt to decide the most appropriate algorithm based on the values passed to fit method. Defaults to "auto". Note: fitting on sparse input will override the setting of this parameter, using brute force.
- **leaf_size** (*int*) – Leaf size passed to BallTree or KDTree. This can affect the speed of the construction and query, as well as the memory required to store the tree. The optimal value depends on the nature of the problem. Defaults to 30.
- **p** (*int*) – Power parameter for the Minkowski metric. When p = 1, this is equivalent to using manhattan_distance (l1), and euclidean_distance (l2) for p = 2. For arbitrary p, minkowski_distance (l_p) is used. Defaults to 2.
- **random_seed** (*int*) – Seed for the random number generator. Defaults to 0.

**Attributes**

| | |
|---|---|
| **hyper-parameter_ranges** | { "n_neighbors": Integer(2, 12), "weights": ["uniform", "distance"], "algorithm": ["auto", "ball_tree", "kd_tree", "brute"], "leaf_size": Integer(10, 30), "p": Integer(1, 5),} |
| **model_family** | ModelFamily.K_NEIGHBORS |
| **modifies_features** | True |
| **modifies_target** | False |
| **name** | KNN Classifier |
| **supported_problem_types** | [ ProblemTypes.BINARY, ProblemTypes.MULTICLASS, ProblemTypes.TIME_SERIES_BINARY, ProblemTypes.TIME_SERIES_MULTICLASS,] |
| **training_only** | False |

**Methods**

| | |
|---|---|
| *clone* | Constructs a new component with the same parameters and random state. |
| *default_parameters* | Returns the default parameters for this component. |
| *describe* | Describe a component and its parameters. |
| *feature_importance* | Returns array of 0's matching the input number of features as feature_importance is not defined for KNN classifiers. |
| *fit* | Fits estimator to data. |
| *get_prediction_intervals* | Find the prediction intervals using the fitted regressor. |
| *load* | Loads component at file path. |
| *needs_fitting* | Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances. |
| *parameters* | Returns the parameters which were used to initialize the component. |
| *predict* | Make predictions using selected features. |
| *predict_proba* | Make probability estimates for labels. |
| *save* | Saves component at file path. |
| *update_parameters* | Updates the parameter dictionary of the component. |

**clone**(*self*)

Constructs a new component with the same parameters and random state.

**Returns** A new instance of this component with identical parameters and random state.

**default_parameters**(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

**Returns** Default parameters for this component.

**Return type** dict

**describe**(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

**Parameters**

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description as dictionary in the format {"name": name, "parameters": parameters}

**Returns** Returns dictionary if return_dict is True, else None.

**Return type** None or dict

**property feature_importance**(*self*)

Returns array of 0's matching the input number of features as feature_importance is not defined for KNN classifiers.

**fit**(*self, X: pandas.DataFrame, y: Optional[pandas.Series] = None*)

Fits estimator to data.

**Parameters**

- **X** (*pd.DataFrame*) – The input training data of shape [n_samples, n_features].
- **y** (*pd.Series, optional*) – The target training data of length [n_samples].

**Returns** self

**get_prediction_intervals**(*self, X: pandas.DataFrame, y: Optional[pandas.Series] = None, coverage: List[float] = None, predictions: pandas.Series = None*) → Dict[str, pandas.Series]

Find the prediction intervals using the fitted regressor.

This function takes the predictions of the fitted estimator and calculates the rolling standard deviation across all predictions using a window size of 5. The lower and upper predictions are determined by taking the percent point (quantile) function of the lower tail probability at each bound multiplied by the rolling standard deviation.

**Parameters**

- **X** (*pd.DataFrame*) – Data of shape [n_samples, n_features].
- **y** (*pd.Series*) – Target data. Ignored.
- **coverage** (*list[float]*) – A list of floats between the values 0 and 1 that the upper and lower bounds of the prediction interval should be calculated for.
- **predictions** (*pd.Series*) – Optional list of predictions to use. If None, will generate predictions using *X*.

**Returns** Prediction intervals, keys are in the format {coverage}_lower or {coverage}_upper.

**Return type** dict

**Raises** **MethodPropertyNotFoundError** – If the estimator does not support Time Series Regression as a problem type.

**static load**(*file_path*)

Loads component at file path.

**Parameters** **file_path** (*str*) – Location to load file.

**Returns** ComponentBase object

**needs_fitting**(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need to be fit or whose fit methods do nothing.

**Returns** True.

**property parameters**(*self*)

Returns the parameters which were used to initialize the component.

**predict**(*self*, *X: pandas.DataFrame*) → pandas.Series

Make predictions using selected features.

**Parameters** **X** (*pd.DataFrame*) – Data of shape [n_samples, n_features].

**Returns** Predicted values.

**Return type** pd.Series

**Raises** **MethodPropertyNotFoundError** – If estimator does not have a predict method or a component_obj that implements predict.

**predict_proba**(*self*, *X: pandas.DataFrame*) → pandas.Series

Make probability estimates for labels.

**Parameters** **X** (*pd.DataFrame*) – Features.

**Returns** Probability estimates.

**Return type** pd.Series

**Raises** **MethodPropertyNotFoundError** – If estimator does not have a predict_proba method or a component_obj that implements predict_proba.

**save**(*self*, *file_path*, *pickle_protocol=cloudpickle.DEFAULT_PROTOCOL*)

Saves component at file path.

**Parameters**

- **file_path** (*str*) – Location to save file.
- **pickle_protocol** (*int*) – The pickle data stream format.

**update_parameters**(*self*, *update_dict*, *reset_fit=True*)

Updates the parameter dictionary of the component.

**Parameters**

- **update_dict** (*dict*) – A dict of parameters to update.
- **reset_fit** (*bool, optional*) – If True, will set *_is_fitted* to False.

**class** evalml.pipelines.components.**LabelEncoder**(*positive_label=None*, *random_seed=0*, ***kwargs*)

A transformer that encodes target labels using values between 0 and num_classes - 1.

**Parameters**

- **positive_label** (*int, str*) – The label for the class that should be treated as positive (1) for binary classification problems. Ignored for multiclass problems. Defaults to None.
- **random_seed** (*int*) – Seed for the random number generator. Defaults to 0. Ignored.

**Attributes**

| | |
|---|---|
| **hyper-parameter_ranges** | {} |
| **modifies_features** | False |
| **modifies_target** | True |
| **name** | Label Encoder |
| **training_only** | False |

**Methods**

| | |
|---|---|
| *clone* | Constructs a new component with the same parameters and random state. |
| *default_parameters* | Returns the default parameters for this component. |
| *describe* | Describe a component and its parameters. |
| *fit* | Fits the label encoder. |
| *fit_transform* | Fit and transform data using the label encoder. |
| *inverse_transform* | Decodes the target data. |
| *load* | Loads component at file path. |
| *needs_fitting* | Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances. |
| *parameters* | Returns the parameters which were used to initialize the component. |
| *save* | Saves component at file path. |
| *transform* | Transform the target using the fitted label encoder. |
| *update_parameters* | Updates the parameter dictionary of the component. |

**clone**(*self*)

Constructs a new component with the same parameters and random state.

**Returns** A new instance of this component with identical parameters and random state.

**default_parameters**(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

**Returns** Default parameters for this component.

**Return type** dict

**describe**(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

**Parameters**

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description as dictionary in the format {"name": name, "parameters": parameters}

**Returns** Returns dictionary if return_dict is True, else None.

**Return type** None or dict

**fit**(*self*, *X*, *y*)

Fits the label encoder.

**Parameters**

- **X** (*pd.DataFrame*) – The input training data of shape [n_samples, n_features]. Ignored.
- **y** (*pd.Series*) – The target training data of length [n_samples].

**Returns** self

**Raises** **ValueError** – If input *y* is None.

**fit_transform**(*self*, *X*, *y*)

Fit and transform data using the label encoder.

**Parameters**

- **X** (*pd.DataFrame*) – The input training data of shape [n_samples, n_features].
- **y** (*pd.Series*) – The target training data of length [n_samples].

**Returns** The original features and an encoded version of the target.

**Return type** pd.DataFrame, pd.Series

**inverse_transform**(*self*, *y*)

Decodes the target data.

**Parameters** **y** (*pd.Series*) – Target data.

**Returns** The decoded version of the target.

**Return type** pd.Series

**Raises** **ValueError** – If input *y* is None.

**static load**(*file_path*)

Loads component at file path.

**Parameters** **file_path** (*str*) – Location to load file.

**Returns** ComponentBase object

**needs_fitting**(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need to be fit or whose fit methods do nothing.

**Returns** True.

**property parameters**(*self*)

Returns the parameters which were used to initialize the component.

**save**(*self*, *file_path*, *pickle_protocol=cloudpickle.DEFAULT_PROTOCOL*)

Saves component at file path.

**Parameters**

- **file_path** (*str*) – Location to save file.
- **pickle_protocol** (*int*) – The pickle data stream format.

**transform**(*self*, *X*, *y=None*)

Transform the target using the fitted label encoder.

**Parameters**

- **X** (*pd.DataFrame*) – The input training data of shape [n_samples, n_features]. Ignored.
- **y** (*pd.Series*) – The target training data of length [n_samples].

**Returns** The original features and an encoded version of the target.

**Return type** pd.DataFrame, pd.Series

**Raises** **ValueError** – If input *y* is None.

**update_parameters**(*self*, *update_dict*, *reset_fit=True*)

Updates the parameter dictionary of the component.

**Parameters**

- **update_dict** (*dict*) – A dict of parameters to update.
- **reset_fit** (*bool, optional*) – If True, will set *_is_fitted* to False.

**class** evalml.pipelines.components.**LightGBMClassifier**(*boosting_type='gbdt'*, *learning_rate=0.1*, *n_estimators=100*, *max_depth=0*, *num_leaves=31*, *min_child_samples=20*, *bagging_fraction=0.9*, *bagging_freq=0*, *n_jobs=- 1*, *random_seed=0*, ***kwargs*)

LightGBM Classifier.

**Parameters**

- **boosting_type** (*string*) – Type of boosting to use. Defaults to "gbdt". - 'gbdt' uses traditional Gradient Boosting Decision Tree - "dart", uses Dropouts meet Multiple Additive Regression Trees - "goss", uses Gradient-based One-Side Sampling - "rf", uses Random Forest
- **learning_rate** (*float*) – Boosting learning rate. Defaults to 0.1.
- **n_estimators** (*int*) – Number of boosted trees to fit. Defaults to 100.
- **max_depth** (*int*) – Maximum tree depth for base learners, <=0 means no limit. Defaults to 0.
- **num_leaves** (*int*) – Maximum tree leaves for base learners. Defaults to 31.
- **min_child_samples** (*int*) – Minimum number of data needed in a child (leaf). Defaults to 20.
- **bagging_fraction** (*float*) – LightGBM will randomly select a subset of features on each iteration (tree) without resampling if this is smaller than 1.0. For example, if set to 0.8, LightGBM will select 80% of features before training each tree. This can be used to speed up training and deal with overfitting. Defaults to 0.9.
- **bagging_freq** (*int*) – Frequency for bagging. 0 means bagging is disabled. k means perform bagging at every k iteration. Every k-th iteration, LightGBM will randomly select bagging_fraction * 100 % of the data to use for the next k iterations. Defaults to 0.
- **n_jobs** (*int or None*) – Number of threads to run in parallel. -1 uses all threads. Defaults to -1.
- **random_seed** (*int*) – Seed for the random number generator. Defaults to 0.

**Attributes**

| | |
|---|---|
| **hyper-parameter_ranges** | { "learning_rate": Real(0.000001, 1), "boosting_type": ["gbdt", "dart", "goss", "rf"], "n_estimators": Integer(10, 100), "max_depth": Integer(0, 10), "num_leaves": Integer(2, 100), "min_child_samples": Integer(1, 100), "bagging_fraction": Real(0.000001, 1), "bagging_freq": Integer(0, 1),} |
| **model_family** | ModelFamily.LIGHTGBM |
| **modifies_features** | True |
| **modifies_target** | False |
| **name** | LightGBM Classifier |
| **SEED_MAX** | SEED_BOUNDS.max_bound |
| **SEED_MIN** | 0 |
| **supported_problem_types** | [ ProblemTypes.BINARY, ProblemTypes.MULTICLASS, ProblemTypes.TIME_SERIES_BINARY, ProblemTypes.TIME_SERIES_MULTICLASS,] |
| **training_only** | False |

**Methods**

| | |
|---|---|
| *clone* | Constructs a new component with the same parameters and random state. |
| *default_parameters* | Returns the default parameters for this component. |
| *describe* | Describe a component and its parameters. |
| *feature_importance* | Returns importance associated with each feature. |
| *fit* | Fits LightGBM classifier component to data. |
| *get_prediction_intervals* | Find the prediction intervals using the fitted regressor. |
| *load* | Loads component at file path. |
| *needs_fitting* | Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances. |
| *parameters* | Returns the parameters which were used to initialize the component. |
| *predict* | Make predictions using the fitted LightGBM classifier. |
| *predict_proba* | Make prediction probabilities using the fitted LightGBM classifier. |
| *save* | Saves component at file path. |
| *update_parameters* | Updates the parameter dictionary of the component. |

**clone**(*self*)

Constructs a new component with the same parameters and random state.

**Returns** A new instance of this component with identical parameters and random state.

**default_parameters**(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

**Returns** Default parameters for this component.

**Return type** dict

**describe**(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

**Parameters**

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description as dictionary in the format {"name": name, "parameters": parameters}

**Returns** Returns dictionary if return_dict is True, else None.

**Return type** None or dict

**property feature_importance**(*self*) → pandas.Series

Returns importance associated with each feature.

**Returns** Importance associated with each feature.

**Return type** np.ndarray

**Raises** **MethodPropertyNotFoundError** – If estimator does not have a feature_importance method or a component_obj that implements feature_importance.

**fit**(*self*, *X*, *y=None*)

Fits LightGBM classifier component to data.

**Parameters**

- **X** (*pd.DataFrame*) – The input training data of shape [n_samples, n_features].
- **y** (*pd.Series*) – The target training data of length [n_samples].

**Returns** self

**get_prediction_intervals**(*self*, *X: pandas.DataFrame*, *y: Optional[pandas.Series] = None*, *coverage: List[float] = None*, *predictions: pandas.Series = None*) → Dict[str, pandas.Series]

Find the prediction intervals using the fitted regressor.

This function takes the predictions of the fitted estimator and calculates the rolling standard deviation across all predictions using a window size of 5. The lower and upper predictions are determined by taking the percent point (quantile) function of the lower tail probability at each bound multiplied by the rolling standard deviation.

**Parameters**

- **X** (*pd.DataFrame*) – Data of shape [n_samples, n_features].
- **y** (*pd.Series*) – Target data. Ignored.
- **coverage** (*list[float]*) – A list of floats between the values 0 and 1 that the upper and lower bounds of the prediction interval should be calculated for.
- **predictions** (*pd.Series*) – Optional list of predictions to use. If None, will generate predictions using *X*.

**Returns** Prediction intervals, keys are in the format {coverage}_lower or {coverage}_upper.

**Return type** dict

**Raises** **MethodPropertyNotFoundError** – If the estimator does not support Time Series Regression as a problem type.

**static load**(*file_path*)

Loads component at file path.

**Parameters** **file_path** (*str*) – Location to load file.

**Returns** ComponentBase object

**needs_fitting**(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need to be fit or whose fit methods do nothing.

**Returns** True.

**property parameters**(*self*)

Returns the parameters which were used to initialize the component.

**predict**(*self*, *X*)

Make predictions using the fitted LightGBM classifier.

**Parameters** **X** (*pd.DataFrame*) – Data of shape [n_samples, n_features].

**Returns** Predicted values.

**Return type** pd.DataFrame

**predict_proba**(*self*, *X*)

Make prediction probabilities using the fitted LightGBM classifier.

**Parameters** **X** (*pd.DataFrame*) – Data of shape [n_samples, n_features].

**Returns** Predicted probability values.

**Return type** pd.DataFrame

**save**(*self*, *file_path*, *pickle_protocol=cloudpickle.DEFAULT_PROTOCOL*)

Saves component at file path.

**Parameters**

- **file_path** (*str*) – Location to save file.
- **pickle_protocol** (*int*) – The pickle data stream format.

**update_parameters**(*self*, *update_dict*, *reset_fit=True*)

Updates the parameter dictionary of the component.

**Parameters**

- **update_dict** (*dict*) – A dict of parameters to update.
- **reset_fit** (*bool, optional*) – If True, will set *_is_fitted* to False.

**class** evalml.pipelines.components.**LightGBMRegressor**(*boosting_type='gbdt'*, *learning_rate=0.1*, *n_estimators=20*, *max_depth=0*, *num_leaves=31*, *min_child_samples=20*, *bagging_fraction=0.9*, *bagging_freq=0*, *n_jobs=- 1*, *random_seed=0*, ***kwargs*)

LightGBM Regressor.

**Parameters**

- **boosting_type** (*string*) – Type of boosting to use. Defaults to "gbdt". - 'gbdt' uses traditional Gradient Boosting Decision Tree - "dart", uses Dropouts meet Multiple Additive Regression Trees - "goss", uses Gradient-based One-Side Sampling - "rf", uses Random Forest
- **learning_rate** (*float*) – Boosting learning rate. Defaults to 0.1.
- **n_estimators** (*int*) – Number of boosted trees to fit. Defaults to 100.
- **max_depth** (*int*) – Maximum tree depth for base learners, <=0 means no limit. Defaults to 0.
- **num_leaves** (*int*) – Maximum tree leaves for base learners. Defaults to 31.
- **min_child_samples** (*int*) – Minimum number of data needed in a child (leaf). Defaults to 20.
- **bagging_fraction** (*float*) – LightGBM will randomly select a subset of features on each iteration (tree) without resampling if this is smaller than 1.0. For example, if set to 0.8, LightGBM will select 80% of features before training each tree. This can be used to speed up training and deal with overfitting. Defaults to 0.9.
- **bagging_freq** (*int*) – Frequency for bagging. 0 means bagging is disabled. k means perform bagging at every k iteration. Every k-th iteration, LightGBM will randomly select bagging_fraction * 100 % of the data to use for the next k iterations. Defaults to 0.
- **n_jobs** (*int or None*) – Number of threads to run in parallel. -1 uses all threads. Defaults to -1.
- **random_seed** (*int*) – Seed for the random number generator. Defaults to 0.

**Attributes**

| | |
|---|---|
| **hyper-parameter_ranges** | { "learning_rate": Real(0.000001, 1), "boosting_type": ["gbdt", "dart", "goss", "rf"], "n_estimators": Integer(10, 100), "max_depth": Integer(0, 10), "num_leaves": Integer(2, 100), "min_child_samples": Integer(1, 100), "bagging_fraction": Real(0.000001, 1), "bagging_freq": Integer(0, 1),} |
| **model_family** | ModelFamily.LIGHTGBM |
| **modifies_features** | True |
| **modifies_target** | False |
| **name** | LightGBM Regressor |
| **SEED_MAX** | SEED_BOUNDS.max_bound |
| **SEED_MIN** | 0 |
| **supported_problem_types** | [ProblemTypes.REGRESSION] |
| **training_only** | False |

**Methods**

| | |
|---|---|
| *clone* | Constructs a new component with the same parameters and random state. |
| *default_parameters* | Returns the default parameters for this component. |
| *describe* | Describe a component and its parameters. |
| *feature_importance* | Returns importance associated with each feature. |
| *fit* | Fits LightGBM regressor to data. |
| *get_prediction_intervals* | Find the prediction intervals using the fitted regressor. |
| *load* | Loads component at file path. |
| *needs_fitting* | Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances. |
| *parameters* | Returns the parameters which were used to initialize the component. |
| *predict* | Make predictions using fitted LightGBM regressor. |
| *predict_proba* | Make probability estimates for labels. |
| *save* | Saves component at file path. |
| *update_parameters* | Updates the parameter dictionary of the component. |

**clone**(*self*)

Constructs a new component with the same parameters and random state.

**Returns** A new instance of this component with identical parameters and random state.

**default_parameters**(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

**Returns** Default parameters for this component.

**Return type** dict

**describe**(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

**Parameters**

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description as dictionary in the format {"name": name, "parameters": parameters}

**Returns** Returns dictionary if return_dict is True, else None.

**Return type** None or dict

**property feature_importance**(*self*) → pandas.Series

Returns importance associated with each feature.

**Returns** Importance associated with each feature.

**Return type** np.ndarray

**Raises** **MethodPropertyNotFoundError** – If estimator does not have a feature_importance method or a component_obj that implements feature_importance.

**fit**(*self*, *X*, *y=None*)

Fits LightGBM regressor to data.

**Parameters**

- **X** (*pd.DataFrame*) – The input training data of shape [n_samples, n_features].
- **y** (*pd.Series*) – The target training data of length [n_samples].

**Returns** self

**get_prediction_intervals**(*self, X: pandas.DataFrame, y: Optional[pandas.Series] = None, coverage: List[float] = None, predictions: pandas.Series = None*) → Dict[str, pandas.Series]

Find the prediction intervals using the fitted regressor.

This function takes the predictions of the fitted estimator and calculates the rolling standard deviation across all predictions using a window size of 5. The lower and upper predictions are determined by taking the percent point (quantile) function of the lower tail probability at each bound multiplied by the rolling standard deviation.

**Parameters**

- **X** (*pd.DataFrame*) – Data of shape [n_samples, n_features].
- **y** (*pd.Series*) – Target data. Ignored.
- **coverage** (*list[float]*) – A list of floats between the values 0 and 1 that the upper and lower bounds of the prediction interval should be calculated for.
- **predictions** (*pd.Series*) – Optional list of predictions to use. If None, will generate predictions using *X*.

**Returns** Prediction intervals, keys are in the format {coverage}_lower or {coverage}_upper.

**Return type** dict

**Raises** **MethodPropertyNotFoundError** – If the estimator does not support Time Series Regression as a problem type.

**static load**(*file_path*)

Loads component at file path.

**Parameters** **file_path** (*str*) – Location to load file.

**Returns** ComponentBase object

**needs_fitting**(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need to be fit or whose fit methods do nothing.

**Returns** True.

**property parameters**(*self*)

Returns the parameters which were used to initialize the component.

**predict**(*self, X*)

Make predictions using fitted LightGBM regressor.

**Parameters** **X** (*pd.DataFrame*) – Data of shape [n_samples, n_features].

**Returns** Predicted values.

**Return type** pd.Series

**predict_proba**(*self*, *X: pandas.DataFrame*) → pandas.Series

Make probability estimates for labels.

**Parameters** **X** (*pd.DataFrame*) – Features.

**Returns** Probability estimates.

**Return type** pd.Series

**Raises** **MethodPropertyNotFoundError** – If estimator does not have a predict_proba method or a component_obj that implements predict_proba.

**save**(*self*, *file_path*, *pickle_protocol=cloudpickle.DEFAULT_PROTOCOL*)

Saves component at file path.

**Parameters**

- **file_path** (*str*) – Location to save file.
- **pickle_protocol** (*int*) – The pickle data stream format.

**update_parameters**(*self*, *update_dict*, *reset_fit=True*)

Updates the parameter dictionary of the component.

**Parameters**

- **update_dict** (*dict*) – A dict of parameters to update.
- **reset_fit** (*bool, optional*) – If True, will set *_is_fitted* to False.

**class** evalml.pipelines.components.**LinearDiscriminantAnalysis**(*n_components=None*, *random_seed=0*, ***kwargs*)

Reduces the number of features by using Linear Discriminant Analysis.

**Parameters**

- **n_components** (*int*) – The number of features to maintain after computation. Defaults to None.
- **random_seed** (*int*) – Seed for the random number generator. Defaults to 0.

**Attributes**

| | |
|---|---|
| **hyper-parameter_ranges** | {} |
| **modifies_features** | True |
| **modifies_target** | False |
| **name** | Linear Discriminant Analysis Transformer |
| **training_only** | False |

**Methods**

| | |
|---|---|
| *clone* | Constructs a new component with the same parameters and random state. |
| *default_parameters* | Returns the default parameters for this component. |
| *describe* | Describe a component and its parameters. |
| *fit* | Fits the LDA component. |
| *fit_transform* | Fit and transform data using the LDA component. |
| *load* | Loads component at file path. |
| *needs_fitting* | Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances. |
| *parameters* | Returns the parameters which were used to initialize the component. |
| *save* | Saves component at file path. |
| *transform* | Transform data using the fitted LDA component. |
| *update_parameters* | Updates the parameter dictionary of the component. |

**clone**(*self*)

Constructs a new component with the same parameters and random state.

**Returns** A new instance of this component with identical parameters and random state.

**default_parameters**(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

**Returns** Default parameters for this component.

**Return type** dict

**describe**(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

**Parameters**

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description as dictionary in the format {"name": name, "parameters": parameters}

**Returns** Returns dictionary if return_dict is True, else None.

**Return type** None or dict

**fit**(*self*, *X*, *y*)

Fits the LDA component.

**Parameters**

- **X** (*pd.DataFrame*) – The input training data of shape [n_samples, n_features].
- **y** (*pd.Series, optional*) – The target training data of length [n_samples].

**Returns** self

**Raises** **ValueError** – If input data is not all numeric.

**fit_transform**(*self*, *X*, *y=None*)

Fit and transform data using the LDA component.

**Parameters**

- **X** (*pd.DataFrame*) – The input training data of shape [n_samples, n_features].
- **y** (*pd.Series, optional*) – The target training data of length [n_samples].

**Returns** Transformed data.

**Return type** pd.DataFrame

**Raises** **ValueError** – If input data is not all numeric.

**static load**(*file_path*)

Loads component at file path.

**Parameters** **file_path** (*str*) – Location to load file.

**Returns** ComponentBase object

**needs_fitting**(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need to be fit or whose fit methods do nothing.

**Returns** True.

**property parameters**(*self*)

Returns the parameters which were used to initialize the component.

**save**(*self*, *file_path*, *pickle_protocol=cloudpickle.DEFAULT_PROTOCOL*)

Saves component at file path.

**Parameters**

- **file_path** (*str*) – Location to save file.
- **pickle_protocol** (*int*) – The pickle data stream format.

**transform**(*self*, *X*, *y=None*)

Transform data using the fitted LDA component.

**Parameters**

- **X** (*pd.DataFrame*) – The input training data of shape [n_samples, n_features].
- **y** (*pd.Series, optional*) – The target training data of length [n_samples].

**Returns** Transformed data.

**Return type** pd.DataFrame

**Raises** **ValueError** – If input data is not all numeric.

**update_parameters**(*self*, *update_dict*, *reset_fit=True*)

Updates the parameter dictionary of the component.

**Parameters**

- **update_dict** (*dict*) – A dict of parameters to update.
- **reset_fit** (*bool, optional*) – If True, will set *_is_fitted* to False.

**class** evalml.pipelines.components.**LinearRegressor**(*fit_intercept=True*, *n_jobs=- 1*, *random_seed=0*, ***kwargs*)

Linear Regressor.

**Parameters**

- **fit_intercept** (*boolean*) – Whether to calculate the intercept for this model. If set to False, no intercept will be used in calculations (i.e. data is expected to be centered). Defaults to True.
- **n_jobs** (*int or None*) – Number of jobs to run in parallel. -1 uses all threads. Defaults to -1.
- **random_seed** (*int*) – Seed for the random number generator. Defaults to 0.

**Attributes**

| | |
|---|---|
| **hyper-parameter_ranges** | { "fit_intercept": [True, False],} |
| **model_family** | ModelFamily.LINEAR_MODEL |
| **modifies_features** | True |
| **modifies_target** | False |
| **name** | Linear Regressor |
| **supported_problem_types** | [ ProblemTypes.REGRESSION, ProblemTypes.TIME_SERIES_REGRESSION,] |
| **training_only** | False |

**Methods**

| | |
|---|---|
| *clone* | Constructs a new component with the same parameters and random state. |
| *default_parameters* | Returns the default parameters for this component. |
| *describe* | Describe a component and its parameters. |
| *feature_importance* | Feature importance for fitted linear regressor. |
| *fit* | Fits estimator to data. |
| *get_prediction_intervals* | Find the prediction intervals using the fitted regressor. |
| *load* | Loads component at file path. |
| *needs_fitting* | Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances. |
| *parameters* | Returns the parameters which were used to initialize the component. |
| *predict* | Make predictions using selected features. |
| *predict_proba* | Make probability estimates for labels. |
| *save* | Saves component at file path. |
| *update_parameters* | Updates the parameter dictionary of the component. |

**clone**(*self*)

Constructs a new component with the same parameters and random state.

**Returns** A new instance of this component with identical parameters and random state.

**default_parameters**(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

**Returns** Default parameters for this component.

**Return type** dict

`describe`(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

**Parameters**

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description as dictionary in the format {"name": name, "parameters": parameters}

**Returns** Returns dictionary if return_dict is True, else None.

**Return type** None or dict

`property feature_importance`(*self*)

Feature importance for fitted linear regressor.

`fit`(*self*, *X: pandas.DataFrame*, *y: Optional[pandas.Series] = None*)

Fits estimator to data.

**Parameters**

- **X** (*pd.DataFrame*) – The input training data of shape [n_samples, n_features].
- **y** (*pd.Series, optional*) – The target training data of length [n_samples].

**Returns** self

`get_prediction_intervals`(*self*, *X: pandas.DataFrame*, *y: Optional[pandas.Series] = None*, *coverage: List[float] = None*, *predictions: pandas.Series = None*) → Dict[str, pandas.Series]

Find the prediction intervals using the fitted regressor.

This function takes the predictions of the fitted estimator and calculates the rolling standard deviation across all predictions using a window size of 5. The lower and upper predictions are determined by taking the percent point (quantile) function of the lower tail probability at each bound multiplied by the rolling standard deviation.

**Parameters**

- **X** (*pd.DataFrame*) – Data of shape [n_samples, n_features].
- **y** (*pd.Series*) – Target data. Ignored.
- **coverage** (*list[float]*) – A list of floats between the values 0 and 1 that the upper and lower bounds of the prediction interval should be calculated for.
- **predictions** (*pd.Series*) – Optional list of predictions to use. If None, will generate predictions using *X*.

**Returns** Prediction intervals, keys are in the format {coverage}_lower or {coverage}_upper.

**Return type** dict

**Raises** **MethodPropertyNotFoundError** – If the estimator does not support Time Series Regression as a problem type.

`static load`(*file_path*)

Loads component at file path.

**Parameters** **file_path** (*str*) – Location to load file.

**Returns** ComponentBase object

**needs_fitting**(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need to be fit or whose fit methods do nothing.

**Returns** True.

**property parameters**(*self*)

Returns the parameters which were used to initialize the component.

**predict**(*self*, *X: pandas.DataFrame*) → pandas.Series

Make predictions using selected features.

**Parameters** **X** (*pd.DataFrame*) – Data of shape [n_samples, n_features].

**Returns** Predicted values.

**Return type** pd.Series

**Raises** **MethodPropertyNotFoundError** – If estimator does not have a predict method or a component_obj that implements predict.

**predict_proba**(*self*, *X: pandas.DataFrame*) → pandas.Series

Make probability estimates for labels.

**Parameters** **X** (*pd.DataFrame*) – Features.

**Returns** Probability estimates.

**Return type** pd.Series

**Raises** **MethodPropertyNotFoundError** – If estimator does not have a predict_proba method or a component_obj that implements predict_proba.

**save**(*self*, *file_path*, *pickle_protocol=cloudpickle.DEFAULT_PROTOCOL*)

Saves component at file path.

**Parameters**

- **file_path** (*str*) – Location to save file.
- **pickle_protocol** (*int*) – The pickle data stream format.

**update_parameters**(*self*, *update_dict*, *reset_fit=True*)

Updates the parameter dictionary of the component.

**Parameters**

- **update_dict** (*dict*) – A dict of parameters to update.
- **reset_fit** (*bool, optional*) – If True, will set *_is_fitted* to False.

**class** evalml.pipelines.components.**LogisticRegressionClassifier**(*penalty='l2'*, *C=1.0*, *multi_class='auto'*, *solver='lbfgs'*, *n_jobs=- 1*, *random_seed=0*, ***kwargs*)

Logistic Regression Classifier.

**Parameters**

- **penalty** (*{"l1", "l2", "elasticnet", "none"}*) – The norm used in penalization. Defaults to "l2".

- **C** (*float*) – Inverse of regularization strength. Must be a positive float. Defaults to 1.0.
- **multi_class** (*{"auto", "ovr", "multinomial"}*) – If the option chosen is "ovr", then a binary problem is fit for each label. For "multinomial" the loss minimised is the multinomial loss fit across the entire probability distribution, even when the data is binary. "multinomial" is unavailable when solver="liblinear". "auto" selects "ovr" if the data is binary, or if solver="liblinear", and otherwise selects "multinomial". Defaults to "auto".
- **solver** (*{"newton-cg", "lbfgs", "liblinear", "sag", "saga"}*) – Algorithm to use in the optimization problem. For small datasets, "liblinear" is a good choice, whereas "sag" and "saga" are faster for large ones. For multiclass problems, only "newton-cg", "sag", "saga" and "lbfgs" handle multinomial loss; "liblinear" is limited to one-versus-rest schemes.
  - "newton-cg", "lbfgs", "sag" and "saga" handle L2 or no penalty
  - "liblinear" and "saga" also handle L1 penalty
  - "saga" also supports "elasticnet" penalty
  - "liblinear" does not support setting penalty='none'

  Defaults to "lbfgs".
- **n_jobs** (*int*) – Number of parallel threads used to run xgboost. Note that creating thread contention will significantly slow down the algorithm. Defaults to -1.
- **random_seed** (*int*) – Seed for the random number generator. Defaults to 0.

**Attributes**

| | |
|---|---|
| **hyper-parameter_ranges** | { "penalty": ["l2"], "C": Real(0.01, 10),} |
| **model_family** | ModelFamily.LINEAR_MODEL |
| **modifies_features** | True |
| **modifies_target** | False |
| **name** | Logistic Regression Classifier |
| **supported_problem_types** | [ ProblemTypes.BINARY, ProblemTypes.MULTICLASS, ProblemTypes.TIME_SERIES_BINARY, ProblemTypes.TIME_SERIES_MULTICLASS,] |
| **training_only** | False |

**Methods**

| | |
|---|---|
| *clone* | Constructs a new component with the same parameters and random state. |
| *default_parameters* | Returns the default parameters for this component. |
| *describe* | Describe a component and its parameters. |
| *feature_importance* | Feature importance for fitted logistic regression classifier. |
| *fit* | Fits estimator to data. |
| *get_prediction_intervals* | Find the prediction intervals using the fitted regressor. |
| *load* | Loads component at file path. |
| *needs_fitting* | Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances. |
| *parameters* | Returns the parameters which were used to initialize the component. |
| *predict* | Make predictions using selected features. |
| *predict_proba* | Make probability estimates for labels. |
| *save* | Saves component at file path. |
| *update_parameters* | Updates the parameter dictionary of the component. |

**clone**(*self*)

Constructs a new component with the same parameters and random state.

**Returns** A new instance of this component with identical parameters and random state.

**default_parameters**(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

**Returns** Default parameters for this component.

**Return type** dict

**describe**(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

**Parameters**

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description as dictionary in the format {"name": name, "parameters": parameters}

**Returns** Returns dictionary if return_dict is True, else None.

**Return type** None or dict

**property feature_importance**(*self*)

Feature importance for fitted logistic regression classifier.

**fit**(*self*, *X: pandas.DataFrame*, *y: Optional[pandas.Series] = None*)

Fits estimator to data.

**Parameters**

- **X** (*pd.DataFrame*) – The input training data of shape [n_samples, n_features].
- **y** (*pd.Series, optional*) – The target training data of length [n_samples].

**Returns** self

**get_prediction_intervals**(*self*, *X: pandas.DataFrame*, *y: Optional[pandas.Series] = None*, *coverage: List[float] = None*, *predictions: pandas.Series = None*) → Dict[str, pandas.Series]

Find the prediction intervals using the fitted regressor.

This function takes the predictions of the fitted estimator and calculates the rolling standard deviation across all predictions using a window size of 5. The lower and upper predictions are determined by taking the percent point (quantile) function of the lower tail probability at each bound multiplied by the rolling standard deviation.

**Parameters**

- **X** (*pd.DataFrame*) – Data of shape [n_samples, n_features].
- **y** (*pd.Series*) – Target data. Ignored.
- **coverage** (*list[float]*) – A list of floats between the values 0 and 1 that the upper and lower bounds of the prediction interval should be calculated for.
- **predictions** (*pd.Series*) – Optional list of predictions to use. If None, will generate predictions using *X*.

**Returns** Prediction intervals, keys are in the format {coverage}_lower or {coverage}_upper.

**Return type** dict

**Raises** **MethodPropertyNotFoundError** – If the estimator does not support Time Series Regression as a problem type.

**static load**(*file_path*)

Loads component at file path.

**Parameters** **file_path** (*str*) – Location to load file.

**Returns** ComponentBase object

**needs_fitting**(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need to be fit or whose fit methods do nothing.

**Returns** True.

**property parameters**(*self*)

Returns the parameters which were used to initialize the component.

**predict**(*self*, *X: pandas.DataFrame*) → pandas.Series

Make predictions using selected features.

**Parameters** **X** (*pd.DataFrame*) – Data of shape [n_samples, n_features].

**Returns** Predicted values.

**Return type** pd.Series

**Raises** **MethodPropertyNotFoundError** – If estimator does not have a predict method or a component_obj that implements predict.

**predict_proba**(*self*, *X: pandas.DataFrame*) → pandas.Series

Make probability estimates for labels.

**Parameters** **X** (*pd.DataFrame*) – Features.

**Returns** Probability estimates.

**Return type** pd.Series

**Raises** **MethodPropertyNotFoundError** – If estimator does not have a predict_proba method or a component_obj that implements predict_proba.

**save**(*self*, *file_path*, *pickle_protocol=cloudpickle.DEFAULT_PROTOCOL*)

Saves component at file path.

**Parameters**

- **file_path** (*str*) – Location to save file.
- **pickle_protocol** (*int*) – The pickle data stream format.

**update_parameters**(*self*, *update_dict*, *reset_fit=True*)

Updates the parameter dictionary of the component.

**Parameters**

- **update_dict** (*dict*) – A dict of parameters to update.
- **reset_fit** (*bool, optional*) – If True, will set *_is_fitted* to False.

**class** evalml.pipelines.components.**LogTransformer**(*random_seed=0*)

Applies a log transformation to the target data.

**Attributes**

| | |
|---|---|
| **hyper-parameter_ranges** | {} |
| **modifies_features** | False |
| **modifies_target** | True |
| **name** | Log Transformer |
| **training_only** | False |

**Methods**

| | |
|---|---|
| *clone* | Constructs a new component with the same parameters and random state. |
| *default_parameters* | Returns the default parameters for this component. |
| *describe* | Describe a component and its parameters. |
| *fit* | Fits the LogTransformer. |
| *fit_transform* | Log transforms the target variable. |
| *inverse_transform* | Apply exponential to target data. |
| *load* | Loads component at file path. |
| *needs_fitting* | Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances. |
| *parameters* | Returns the parameters which were used to initialize the component. |
| *save* | Saves component at file path. |
| *transform* | Log transforms the target variable. |
| *update_parameters* | Updates the parameter dictionary of the component. |

**clone**(*self*)

Constructs a new component with the same parameters and random state.

**Returns** A new instance of this component with identical parameters and random state.

**default_parameters**(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

**Returns** Default parameters for this component.

**Return type** dict

**describe**(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

**Parameters**

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description as dictionary in the format {"name": name, "parameters": parameters}

**Returns** Returns dictionary if return_dict is True, else None.

**Return type** None or dict

**fit**(*self*, *X*, *y=None*)

Fits the LogTransformer.

**Parameters**

- **X** (*pd.DataFrame or np.ndarray*) – Ignored.
- **y** (*pd.Series, optional*) – Ignored.

**Returns** self

**fit_transform**(*self*, *X*, *y=None*)

Log transforms the target variable.

**Parameters**

- **X** (*pd.DataFrame, optional*) – Ignored.
- **y** (*pd.Series*) – Target variable to log transform.

**Returns**

**The input features are returned without modification. The target** variable y is log transformed.

**Return type** tuple of pd.DataFrame, pd.Series

**inverse_transform**(*self*, *y*)

Apply exponential to target data.

**Parameters** **y** (*pd.Series*) – Target variable.

**Returns** Target with exponential applied.

**Return type** pd.Series

**static load**(*file_path*)

Loads component at file path.

**Parameters** **file_path** (*str*) – Location to load file.

**Returns** ComponentBase object

**needs_fitting**(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need to be fit or whose fit methods do nothing.

**Returns** True.

**property parameters**(*self*)

Returns the parameters which were used to initialize the component.

**save**(*self*, *file_path*, *pickle_protocol=cloudpickle.DEFAULT_PROTOCOL*)

Saves component at file path.

**Parameters**

- **file_path** (*str*) – Location to save file.
- **pickle_protocol** (*int*) – The pickle data stream format.

**transform**(*self*, *X*, *y=None*)

Log transforms the target variable.

**Parameters**

- **X** (*pd.DataFrame, optional*) – Ignored.
- **y** (*pd.Series*) – Target data to log transform.

**Returns**

**The input features are returned without modification. The target** variable y is log transformed.

**Return type** tuple of pd.DataFrame, pd.Series

**update_parameters**(*self*, *update_dict*, *reset_fit=True*)

Updates the parameter dictionary of the component.

**Parameters**

- **update_dict** (*dict*) – A dict of parameters to update.
- **reset_fit** (*bool, optional*) – If True, will set *_is_fitted* to False.

**class** evalml.pipelines.components.**LSA**(*random_seed=0*, ***kwargs*)

Transformer to calculate the Latent Semantic Analysis Values of text input.

**Parameters** **random_seed** (*int*) – Seed for the random number generator. Defaults to 0.

**Attributes**

| | |
|---|---|
| **hyper-parameter_ranges** | {} |
| **modifies_features** | True |
| **modifies_target** | False |
| **name** | LSA Transformer |
| **training_only** | False |

**Methods**

| | |
|---|---|
| *clone* | Constructs a new component with the same parameters and random state. |
| *default_parameters* | Returns the default parameters for this component. |
| *describe* | Describe a component and its parameters. |
| *fit* | Fits the input data. |
| *fit_transform* | Fits on X and transforms X. |
| *load* | Loads component at file path. |
| *needs_fitting* | Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances. |
| *parameters* | Returns the parameters which were used to initialize the component. |
| *save* | Saves component at file path. |
| *transform* | Transforms data X by applying the LSA pipeline. |
| *update_parameters* | Updates the parameter dictionary of the component. |

**clone**(*self*)

Constructs a new component with the same parameters and random state.

**Returns** A new instance of this component with identical parameters and random state.

**default_parameters**(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

**Returns** Default parameters for this component.

**Return type** dict

**describe**(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

**Parameters**

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description as dictionary in the format {"name": name, "parameters": parameters}

**Returns** Returns dictionary if return_dict is True, else None.

**Return type** None or dict

**fit**(*self*, *X*, *y=None*)

Fits the input data.

**Parameters**

- **X** (*pd.DataFrame*) – The data to transform.
- **y** (*pd.Series, optional*) – Ignored.

**Returns** self

**fit_transform**(*self*, *X*, *y=None*)

Fits on X and transforms X.

**Parameters**

- **X** (*pd.DataFrame*) – Data to fit and transform.
- **y** (*pd.Series*) – Target data.

**Returns** Transformed X.

**Return type** pd.DataFrame

**Raises** **MethodPropertyNotFoundError** – If transformer does not have a transform method or a component_obj that implements transform.

**static load**(*file_path*)

Loads component at file path.

**Parameters** **file_path** (*str*) – Location to load file.

**Returns** ComponentBase object

**needs_fitting**(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need to be fit or whose fit methods do nothing.

**Returns** True.

**property parameters**(*self*)

Returns the parameters which were used to initialize the component.

**save**(*self*, *file_path*, *pickle_protocol=cloudpickle.DEFAULT_PROTOCOL*)

Saves component at file path.

**Parameters**

- **file_path** (*str*) – Location to save file.
- **pickle_protocol** (*int*) – The pickle data stream format.

**transform**(*self*, *X*, *y=None*)

Transforms data X by applying the LSA pipeline.

**Parameters**

- **X** (*pd.DataFrame*) – The data to transform.
- **y** (*pd.Series, optional*) – Ignored.

**Returns**

**Transformed X. The original column is removed and replaced with two columns of the** format *LSA(original_column_name)[feature_number]*, where *feature_number* is 0 or 1.

**Return type** pd.DataFrame

**update_parameters**(*self*, *update_dict*, *reset_fit=True*)

Updates the parameter dictionary of the component.

**Parameters**

- **update_dict** (*dict*) – A dict of parameters to update.
- **reset_fit** (*bool, optional*) – If True, will set *_is_fitted* to False.

**class** evalml.pipelines.components.**NaturalLanguageFeaturizer**(*random_seed=0*, ***kwargs*)

Transformer that can automatically featurize text columns using featuretools' nlp_primitives.

Since models cannot handle non-numeric data, any text must be broken down into features that provide useful information about that text. This component splits each text column into several informative features: Diversity Score, Mean Characters per Word, Polarity Score, LSA (Latent Semantic Analysis), Number of Characters, and Number of Words. Calling transform on this component will replace any text columns in the given dataset with these numeric columns.

**Parameters** **random_seed** (*int*) – Seed for the random number generator. Defaults to 0.

**Attributes**

| | |
|---|---|
| **hyper-parameter_ranges** | {} |
| **modifies_features** | True |
| **modifies_target** | False |
| **name** | Natural Language Featurizer |
| **training_only** | False |

**Methods**

| | |
|---|---|
| *clone* | Constructs a new component with the same parameters and random state. |
| *default_parameters* | Returns the default parameters for this component. |
| *describe* | Describe a component and its parameters. |
| *fit* | Fits component to data. |
| *fit_transform* | Fits on X and transforms X. |
| *load* | Loads component at file path. |
| *needs_fitting* | Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances. |
| *parameters* | Returns the parameters which were used to initialize the component. |
| *save* | Saves component at file path. |
| *transform* | Transforms data X by creating new features using existing text columns. |
| *update_parameters* | Updates the parameter dictionary of the component. |

**clone**(*self*)

Constructs a new component with the same parameters and random state.

**Returns** A new instance of this component with identical parameters and random state.

**default_parameters**(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

**Returns** Default parameters for this component.

**Return type** dict

**describe**(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

**Parameters**

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description as dictionary in the format {"name": name, "parameters": parameters}

**Returns** Returns dictionary if return_dict is True, else None.

**Return type** None or dict

**fit**(*self*, *X*, *y=None*)

Fits component to data.

**Parameters**

- **X** (*pd.DataFrame or np.ndarray*) – The input training data of shape [n_samples, n_features]
- **y** (*pd.Series*) – The target training data of length [n_samples]

**Returns** self

**fit_transform**(*self*, *X*, *y=None*)

Fits on X and transforms X.

**Parameters**

- **X** (*pd.DataFrame*) – Data to fit and transform.
- **y** (*pd.Series*) – Target data.

**Returns** Transformed X.

**Return type** pd.DataFrame

**Raises** **MethodPropertyNotFoundError** – If transformer does not have a transform method or a component_obj that implements transform.

**static load**(*file_path*)

Loads component at file path.

**Parameters** **file_path** (*str*) – Location to load file.

**Returns** ComponentBase object

**needs_fitting**(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need to be fit or whose fit methods do nothing.

**Returns** True.

**property parameters**(*self*)

Returns the parameters which were used to initialize the component.

**save**(*self*, *file_path*, *pickle_protocol=cloudpickle.DEFAULT_PROTOCOL*)

Saves component at file path.

**Parameters**

- **file_path** (*str*) – Location to save file.
- **pickle_protocol** (*int*) – The pickle data stream format.

**transform**(*self*, *X*, *y=None*)

Transforms data X by creating new features using existing text columns.

**Parameters**

- **X** (*pd.DataFrame*) – The data to transform.
- **y** (*pd.Series, optional*) – Ignored.

**Returns** Transformed X

**Return type** pd.DataFrame

**update_parameters**(*self*, *update_dict*, *reset_fit=True*)

Updates the parameter dictionary of the component.

**Parameters**

- **update_dict** (*dict*) – A dict of parameters to update.
- **reset_fit** (*bool, optional*) – If True, will set *_is_fitted* to False.

**class** evalml.pipelines.components.**OneHotEncoder**(*top_n=10*, *features_to_encode=None*, *categories=None*, *drop='if_binary'*, *handle_unknown='ignore'*, *handle_missing='error'*, *random_seed=0*, ***kwargs*)

A transformer that encodes categorical features in a one-hot numeric array.

**Parameters**

- **top_n** (*int*) – Number of categories per column to encode. If None, all categories will be encoded. Otherwise, the *n* most frequent will be encoded and all others will be dropped. Defaults to 10.
- **features_to_encode** (*list[str]*) – List of columns to encode. All other columns will remain untouched. If None, all appropriate columns will be encoded. Defaults to None.
- **categories** (*list*) – A two dimensional list of categories, where *categories[i]* is a list of the categories for the column at index *i*. This can also be *None*, or *"auto"* if *top_n* is not None. Defaults to None.
- **drop** (*string, list*) – Method ("first" or "if_binary") to use to drop one category per feature. Can also be a list specifying which categories to drop for each feature. Defaults to 'if_binary'.
- **handle_unknown** (*string*) – Whether to ignore or error for unknown categories for a feature encountered during *fit* or *transform*. If either *top_n* or *categories* is used to limit the number of categories per column, this must be "ignore". Defaults to "ignore".
- **handle_missing** (*string*) – Options for how to handle missing (NaN) values encountered during *fit* or *transform*. If this is set to "as_category" and NaN values are within the *n* most frequent, "nan" values will be encoded as their own column. If this is set to "error", any missing values encountered will raise an error. Defaults to "error".
- **random_seed** (*int*) – Seed for the random number generator. Defaults to 0.

**Attributes**

| | |
|---|---|
| **hyper-parameter_ranges** | {} |
| **modifies_features** | True |
| **modifies_target** | False |
| **name** | One Hot Encoder |
| **training_only** | False |

**Methods**

| | |
|---|---|
| categories | Returns a list of the unique categories to be encoded for the particular feature, in order. |
| clone | Constructs a new component with the same parameters and random state. |
| default_parameters | Returns the default parameters for this component. |
| describe | Describe a component and its parameters. |
| fit | Fits the one-hot encoder component. |
| fit_transform | Fits on X and transforms X. |
| get_feature_names | Return feature names for the categorical features after fitting. |
| load | Loads component at file path. |
| needs_fitting | Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances. |
| parameters | Returns the parameters which were used to initialize the component. |
| save | Saves component at file path. |
| transform | One-hot encode the input data. |
| update_parameters | Updates the parameter dictionary of the component. |

**categories**(*self*, *feature_name*)

Returns a list of the unique categories to be encoded for the particular feature, in order.

**Parameters** **feature_name** (*str*) – The name of any feature provided to one-hot encoder during fit.

**Returns** The unique categories, in the same dtype as they were provided during fit.

**Return type** np.ndarray

**Raises** **ValueError** – If feature was not provided to one-hot encoder as a training feature.

**clone**(*self*)

Constructs a new component with the same parameters and random state.

**Returns** A new instance of this component with identical parameters and random state.

**default_parameters**(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

**Returns** Default parameters for this component.

**Return type** dict

**describe**(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

**Parameters**

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description as dictionary in the format {"name": name, "parameters": parameters}

**Returns** Returns dictionary if return_dict is True, else None.

**Return type** None or dict

**fit**(*self*, *X*, *y=None*)

Fits the one-hot encoder component.

**Parameters**

- **X** (*pd.DataFrame*) – The input training data of shape [n_samples, n_features].
- **y** (*pd.Series, optional*) – The target training data of length [n_samples].

**Returns** self

**Raises** **ValueError** – If encoding a column failed.

**fit_transform**(*self*, *X*, *y=None*)

Fits on X and transforms X.

**Parameters**

- **X** (*pd.DataFrame*) – Data to fit and transform.
- **y** (*pd.Series*) – Target data.

**Returns** Transformed X.

**Return type** pd.DataFrame

**Raises** **MethodPropertyNotFoundError** – If transformer does not have a transform method or a component_obj that implements transform.

**get_feature_names**(*self*)

Return feature names for the categorical features after fitting.

Feature names are formatted as {column name}_{category name}. In the event of a duplicate name, an integer will be added at the end of the feature name to distinguish it.

For example, consider a dataframe with a column called "A" and category "x_y" and another column called "A_x" with "y". In this example, the feature names would be "A_x_y" and "A_x_y_1".

**Returns** The feature names after encoding, provided in the same order as input_features.

**Return type** np.ndarray

**static load**(*file_path*)

Loads component at file path.

**Parameters** **file_path** (*str*) – Location to load file.

**Returns** ComponentBase object

**needs_fitting**(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need to be fit or whose fit methods do nothing.

**Returns** True.

**property parameters**(*self*)

Returns the parameters which were used to initialize the component.

**save**(*self*, *file_path*, *pickle_protocol=cloudpickle.DEFAULT_PROTOCOL*)

Saves component at file path.

**Parameters**

- **file_path** (*str*) – Location to save file.

- **pickle_protocol** (*int*) – The pickle data stream format.

**transform**(*self*, *X*, *y=None*)

One-hot encode the input data.

**Parameters**

- **X** (*pd.DataFrame*) – Features to one-hot encode.
- **y** (*pd.Series*) – Ignored.

**Returns** Transformed data, where each categorical feature has been encoded into numerical columns using one-hot encoding.

**Return type** pd.DataFrame

**update_parameters**(*self*, *update_dict*, *reset_fit=True*)

Updates the parameter dictionary of the component.

**Parameters**

- **update_dict** (*dict*) – A dict of parameters to update.
- **reset_fit** (*bool, optional*) – If True, will set *_is_fitted* to False.

**class** evalml.pipelines.components.**OrdinalEncoder**(*features_to_encode=None*, *categories=None*, *handle_unknown='error'*, *unknown_value=None*, *encoded_missing_value=None*, *random_seed=0*, ***kwargs*)

A transformer that encodes ordinal features as an array of ordinal integers representing the relative order of categories.

**Parameters**

- **features_to_encode** (*list[str]*) – List of columns to encode. All other columns will remain untouched. If None, all appropriate columns will be encoded. Defaults to None. The order of columns does not matter.
- **categories** (*dict[str, list[str]]*) – A dictionary mapping column names to their categories in the dataframes passed in at fit and transform. The order of categories specified for a column does not matter. Any category found in the data that is not present in categories will be handled as an unknown value. To not have unknown values raise an error, set handle_unknown to "use_encoded_value". Defaults to None.
- **handle_unknown** (*"error" or "use_encoded_value"*) – Whether to ignore or error for unknown categories for a feature encountered during *fit* or *transform*. When set to "error", an error will be raised when an unknown category is found. When set to "use_encoded_value", unknown categories will be encoded as the value given for the parameter unknown_value. Defaults to "error."
- **unknown_value** (*int or np.nan*) – The value to use for unknown categories seen during fit or transform. Required when the parameter handle_unknown is set to "use_encoded_value." The value has to be distinct from the values used to encode any of the categories in fit. Defaults to None.
- **encoded_missing_value** (*int or np.nan*) – The value to use for missing (null) values seen during fit or transform. Defaults to np.nan.
- **random_seed** (*int*) – Seed for the random number generator. Defaults to 0.

**Attributes**

| | |
|---|---|
| **hyper-parameter_ranges** | {} |
| **modifies_features** | True |
| **modifies_target** | False |
| **name** | Ordinal Encoder |
| **training_only** | False |

**Methods**

| | |
|---|---|
| *categories* | Returns a list of the unique categories to be encoded for the particular feature, in order. |
| *clone* | Constructs a new component with the same parameters and random state. |
| *default_parameters* | Returns the default parameters for this component. |
| *describe* | Describe a component and its parameters. |
| *fit* | Fits the ordinal encoder component. |
| *fit_transform* | Fits on X and transforms X. |
| *get_feature_names* | Return feature names for the ordinal features after fitting. |
| *load* | Loads component at file path. |
| *needs_fitting* | Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances. |
| *parameters* | Returns the parameters which were used to initialize the component. |
| *save* | Saves component at file path. |
| *transform* | Ordinally encode the input data. |
| *update_parameters* | Updates the parameter dictionary of the component. |

**categories**(*self*, *feature_name*)

Returns a list of the unique categories to be encoded for the particular feature, in order.

**Parameters** **feature_name** (*str*) – The name of any feature provided to ordinal encoder during fit.

**Returns** The unique categories, in the same dtype as they were provided during fit.

**Return type** np.ndarray

**Raises** **ValueError** – If feature was not provided to ordinal encoder as a training feature.

**clone**(*self*)

Constructs a new component with the same parameters and random state.

**Returns** A new instance of this component with identical parameters and random state.

**default_parameters**(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

**Returns** Default parameters for this component.

**Return type** dict

**describe**(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

**Parameters**

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description as dictionary in the format {"name": name, "parameters": parameters}

**Returns** Returns dictionary if return_dict is True, else None.

**Return type** None or dict

**fit**(*self*, *X*, *y=None*)

Fits the ordinal encoder component.

**Parameters**

- **X** (*pd.DataFrame*) – The input training data of shape [n_samples, n_features].
- **y** (*pd.Series, optional*) – The target training data of length [n_samples].

**Returns** self

**Raises**

- **ValueError** – If encoding a column failed.
- **TypeError** – If non-Ordinal columns are specified in features_to_encode.

**fit_transform**(*self*, *X*, *y=None*)

Fits on X and transforms X.

**Parameters**

- **X** (*pd.DataFrame*) – Data to fit and transform.
- **y** (*pd.Series*) – Target data.

**Returns** Transformed X.

**Return type** pd.DataFrame

**Raises** **MethodPropertyNotFoundError** – If transformer does not have a transform method or a component_obj that implements transform.

**get_feature_names**(*self*)

Return feature names for the ordinal features after fitting.

Feature names are formatted as {column name}_ordinal_encoding.

**Returns** The feature names after encoding, provided in the same order as input_features.

**Return type** np.ndarray

**static load**(*file_path*)

Loads component at file path.

**Parameters** **file_path** (*str*) – Location to load file.

**Returns** ComponentBase object

**needs_fitting**(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need to be fit or whose fit methods do nothing.

**Returns** True.

**property parameters**(*self*)

Returns the parameters which were used to initialize the component.

**save**(*self*, *file_path*, *pickle_protocol=cloudpickle.DEFAULT_PROTOCOL*)

Saves component at file path.

**Parameters**

- **file_path** (*str*) – Location to save file.
- **pickle_protocol** (*int*) – The pickle data stream format.

**transform**(*self*, *X*, *y=None*)

Ordinally encode the input data.

**Parameters**

- **X** (*pd.DataFrame*) – Features to encode.
- **y** (*pd.Series*) – Ignored.

**Returns** Transformed data, where each ordinal feature has been encoded into a numerical column where ordinal integers represent the relative order of categories.

**Return type** pd.DataFrame

**update_parameters**(*self*, *update_dict*, *reset_fit=True*)

Updates the parameter dictionary of the component.

**Parameters**

- **update_dict** (*dict*) – A dict of parameters to update.
- **reset_fit** (*bool, optional*) – If True, will set *_is_fitted* to False.

**class** evalml.pipelines.components.**Oversampler**(*sampling_ratio=0.25*, *sampling_ratio_dict=None*, *k_neighbors_default=5*, *n_jobs=- 1*, *random_seed=0*, ***kwargs*)

SMOTE Oversampler component. Will automatically select whether to use SMOTE, SMOTEN, or SMOTENC based on inputs to the component.

**Parameters**

- **sampling_ratio** (*float*) – This is the goal ratio of the minority to majority class, with range (0, 1]. A value of 0.25 means we want a 1:4 ratio of the minority to majority class after oversampling. We will create the a sampling dictionary using this ratio, with the keys corresponding to the class and the values responding to the number of samples. Defaults to 0.25.
- **sampling_ratio_dict** (*dict*) – A dictionary specifying the desired balanced ratio for each target value. For instance, in a binary case where class 1 is the minority, we could specify: *sampling_ratio_dict={0: 0.5, 1: 1}*, which means we would undersample class 0 to have twice the number of samples as class 1 (minority:majority ratio = 0.5), and don't sample class 1. Overrides sampling_ratio if provided. Defaults to None.

- **k_neighbors_default** (*int*) – The number of nearest neighbors used to construct synthetic samples. This is the default value used, but the actual k_neighbors value might be smaller if there are less samples. Defaults to 5.
- **n_jobs** (*int*) – The number of CPU cores to use. Defaults to -1.
- **random_seed** (*int*) – The seed to use for random sampling. Defaults to 0.

**Attributes**

| | |
|---|---|
| **hyper-parameter_ranges** | None |
| **modifies_features** | True |
| **modifies_target** | True |
| **name** | Oversampler |
| **training_only** | True |

**Methods**

| | |
|---|---|
| *clone* | Constructs a new component with the same parameters and random state. |
| *default_parameters* | Returns the default parameters for this component. |
| *describe* | Describe a component and its parameters. |
| *fit* | Fits oversampler to data. |
| *fit_transform* | Fit and transform data using the sampler component. |
| *load* | Loads component at file path. |
| *needs_fitting* | Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances. |
| *parameters* | Returns the parameters which were used to initialize the component. |
| *save* | Saves component at file path. |
| *transform* | Transforms the input data by Oversampling the data. |
| *update_parameters* | Updates the parameter dictionary of the component. |

**clone**(*self*)

Constructs a new component with the same parameters and random state.

**Returns** A new instance of this component with identical parameters and random state.

**default_parameters**(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

**Returns** Default parameters for this component.

**Return type** dict

**describe**(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

**Parameters**

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description as dictionary in the format {"name": name, "parameters": parameters}

**Returns** Returns dictionary if return_dict is True, else None.

**Return type** None or dict

**fit**(*self*, *X*, *y*)

Fits oversampler to data.

**Parameters**

- **X** (*pd.DataFrame*) – The input training data of shape [n_samples, n_features].
- **y** (*pd.Series, optional*) – The target training data of length [n_samples].

**Returns** self

**fit_transform**(*self*, *X*, *y*)

Fit and transform data using the sampler component.

**Parameters**

- **X** (*pd.DataFrame*) – The input training data of shape [n_samples, n_features].
- **y** (*pd.Series, optional*) – The target training data of length [n_samples].

**Returns** Transformed data.

**Return type** (pd.DataFrame, pd.Series)

**static load**(*file_path*)

Loads component at file path.

**Parameters** **file_path** (*str*) – Location to load file.

**Returns** ComponentBase object

**needs_fitting**(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need to be fit or whose fit methods do nothing.

**Returns** True.

**property parameters**(*self*)

Returns the parameters which were used to initialize the component.

**save**(*self*, *file_path*, *pickle_protocol=cloudpickle.DEFAULT_PROTOCOL*)

Saves component at file path.

**Parameters**

- **file_path** (*str*) – Location to save file.
- **pickle_protocol** (*int*) – The pickle data stream format.

**transform**(*self*, *X*, *y=None*)

Transforms the input data by Oversampling the data.

**Parameters**

- **X** (*pd.DataFrame*) – Training features.

- **y** (*pd.Series*) – Target.

**Returns** Transformed features and target.

**Return type** pd.DataFrame, pd.Series

**update_parameters**(*self*, *update_dict*, *reset_fit=True*)

Updates the parameter dictionary of the component.

**Parameters**

- **update_dict** (*dict*) – A dict of parameters to update.
- **reset_fit** (*bool, optional*) – If True, will set *_is_fitted* to False.

**class** evalml.pipelines.components.**PCA**(*variance=0.95*, *n_components=None*, *random_seed=0*, ***kwargs*)

Reduces the number of features by using Principal Component Analysis (PCA).

**Parameters**

- **variance** (*float*) – The percentage of the original data variance that should be preserved when reducing the number of features. Defaults to 0.95.
- **n_components** (*int*) – The number of features to maintain after computing SVD. Defaults to None, but will override variance variable if set.
- **random_seed** (*int*) – Seed for the random number generator. Defaults to 0.

**Attributes**

| | |
|---|---|
| **hyper-parameter_ranges** | Real(0.25, 1)}:type: {"variance" |
| **modifies_features** | True |
| **modifies_target** | False |
| **name** | PCA Transformer |
| **training_only** | False |

**Methods**

| | |
|---|---|
| *clone* | Constructs a new component with the same parameters and random state. |
| *default_parameters* | Returns the default parameters for this component. |
| *describe* | Describe a component and its parameters. |
| *fit* | Fits the PCA component. |
| *fit_transform* | Fit and transform data using the PCA component. |
| *load* | Loads component at file path. |
| *needs_fitting* | Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances. |
| *parameters* | Returns the parameters which were used to initialize the component. |
| *save* | Saves component at file path. |
| *transform* | Transform data using fitted PCA component. |
| *update_parameters* | Updates the parameter dictionary of the component. |

**clone**(*self*)

Constructs a new component with the same parameters and random state.

**Returns** A new instance of this component with identical parameters and random state.

**default_parameters**(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

**Returns** Default parameters for this component.

**Return type** dict

**describe**(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

**Parameters**

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description as dictionary in the format {"name": name, "parameters": parameters}

**Returns** Returns dictionary if return_dict is True, else None.

**Return type** None or dict

**fit**(*self*, *X*, *y=None*)

Fits the PCA component.

**Parameters**

- **X** (*pd.DataFrame*) – The input training data of shape [n_samples, n_features].
- **y** (*pd.Series, optional*) – The target training data of length [n_samples].

**Returns** self

**Raises** **ValueError** – If input data is not all numeric.

**fit_transform**(*self*, *X*, *y=None*)

Fit and transform data using the PCA component.

**Parameters**

- **X** (*pd.DataFrame*) – The input training data of shape [n_samples, n_features].
- **y** (*pd.Series, optional*) – The target training data of length [n_samples].

**Returns** Transformed data.

**Return type** pd.DataFrame

**Raises** **ValueError** – If input data is not all numeric.

**static load**(*file_path*)

Loads component at file path.

**Parameters** **file_path** (*str*) – Location to load file.

**Returns** ComponentBase object

**needs_fitting**(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need to be fit or whose fit methods do nothing.

**Returns** True.

**property parameters**(*self*)

Returns the parameters which were used to initialize the component.

**save**(*self*, *file_path*, *pickle_protocol=cloudpickle.DEFAULT_PROTOCOL*)

Saves component at file path.

**Parameters**

- **file_path** (*str*) – Location to save file.
- **pickle_protocol** (*int*) – The pickle data stream format.

**transform**(*self*, *X*, *y=None*)

Transform data using fitted PCA component.

**Parameters**

- **X** (*pd.DataFrame*) – The input training data of shape [n_samples, n_features].
- **y** (*pd.Series, optional*) – The target training data of length [n_samples].

**Returns** Transformed data.

**Return type** pd.DataFrame

**Raises** **ValueError** – If input data is not all numeric.

**update_parameters**(*self*, *update_dict*, *reset_fit=True*)

Updates the parameter dictionary of the component.

**Parameters**

- **update_dict** (*dict*) – A dict of parameters to update.
- **reset_fit** (*bool, optional*) – If True, will set *_is_fitted* to False.

**class** evalml.pipelines.components.**PerColumnImputer**(*impute_strategies=None*, *random_seed=0*, ***kwargs*)

Imputes missing data according to a specified imputation strategy per column.

**Parameters**

- **impute_strategies** (*dict*) – Column and {"impute_strategy": strategy, "fill_value":value} pairings. Valid values for impute strategy include "mean", "median", "most_frequent", "constant" for numerical data, and "most_frequent", "constant" for object data types. Defaults to None, which uses "most_frequent" for all columns. When impute_strategy == "constant", fill_value is used to replace missing data. When None, uses 0 when imputing numerical data and "missing_value" for strings or object data types.
- **random_seed** (*int*) – Seed for the random number generator. Defaults to 0.

**Attributes**

| | |
|---|---|
| **hyper-parameter_ranges** | {} |
| **modifies_features** | True |
| **modifies_target** | False |
| **name** | Per Column Imputer |
| **training_only** | False |

**Methods**

| | |
|---|---|
| *clone* | Constructs a new component with the same parameters and random state. |
| *default_parameters* | Returns the default parameters for this component. |
| *describe* | Describe a component and its parameters. |
| *fit* | Fits imputers on input data. |
| *fit_transform* | Fits on X and transforms X. |
| *load* | Loads component at file path. |
| *needs_fitting* | Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances. |
| *parameters* | Returns the parameters which were used to initialize the component. |
| *save* | Saves component at file path. |
| *transform* | Transforms input data by imputing missing values. |
| *update_parameters* | Updates the parameter dictionary of the component. |

**clone**(*self*)

Constructs a new component with the same parameters and random state.

**Returns** A new instance of this component with identical parameters and random state.

**default_parameters**(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

**Returns** Default parameters for this component.

**Return type** dict

**describe**(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

**Parameters**

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description as dictionary in the format {"name": name, "parameters": parameters}

**Returns** Returns dictionary if return_dict is True, else None.

**Return type** None or dict

**fit**(*self*, *X*, *y=None*)

Fits imputers on input data.

**Parameters**

- **X** (*pd.DataFrame or np.ndarray*) – The input training data of shape [n_samples, n_features] to fit.
- **y** (*pd.Series, optional*) – The target training data of length [n_samples]. Ignored.

**Returns** self

**fit_transform**(*self*, *X*, *y=None*)

Fits on X and transforms X.

**Parameters**

- **X** (*pd.DataFrame*) – Data to fit and transform.
- **y** (*pd.Series*) – Target data.

**Returns** Transformed X.

**Return type** pd.DataFrame

**Raises** **MethodPropertyNotFoundError** – If transformer does not have a transform method or a component_obj that implements transform.

**static load**(*file_path*)

Loads component at file path.

**Parameters** **file_path** (*str*) – Location to load file.

**Returns** ComponentBase object

**needs_fitting**(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need to be fit or whose fit methods do nothing.

**Returns** True.

**property parameters**(*self*)

Returns the parameters which were used to initialize the component.

**save**(*self*, *file_path*, *pickle_protocol=cloudpickle.DEFAULT_PROTOCOL*)

Saves component at file path.

**Parameters**

- **file_path** (*str*) – Location to save file.
- **pickle_protocol** (*int*) – The pickle data stream format.

**transform**(*self*, *X*, *y=None*)

Transforms input data by imputing missing values.

**Parameters**

- **X** (*pd.DataFrame or np.ndarray*) – The input training data of shape [n_samples, n_features] to transform.
- **y** (*pd.Series, optional*) – The target training data of length [n_samples]. Ignored.

**Returns** Transformed X

**Return type** pd.DataFrame

**update_parameters**(*self*, *update_dict*, *reset_fit=True*)

Updates the parameter dictionary of the component.

**Parameters**

- **update_dict** (*dict*) – A dict of parameters to update.
- **reset_fit** (*bool, optional*) – If True, will set *_is_fitted* to False.

**class** evalml.pipelines.components.**PolynomialDecomposer**(*time_index: str = None*, *degree: int = 1*, *period: int = - 1*, *random_seed: int = 0*, ***kwargs*)

Removes trends and seasonality from time series by fitting a polynomial and moving average to the data.

**Scikit-learn's PolynomialForecaster is used to generate the additive trend portion of the target data. A polynomial** will be fit to the data during fit. That additive polynomial trend will be removed during fit so that statsmodel's seasonal_decompose can determine the addititve seasonality of the data by using rolling averages over the series' inferred periodicity.

For example, daily time series data will generate rolling averages over the first week of data, normalize out the mean and return those 7 averages repeated over the entire length of the given series. Those seven averages, repeated as many times as necessary to match the length of the given target data, will be used as the seasonal signal of the data.

**Parameters**

- **time_index** (*str*) – Specifies the name of the column in X that provides the datetime objects. Defaults to None.
- **degree** (*int*) – Degree for the polynomial. If 1, linear model is fit to the data. If 2, quadratic model is fit, etc. Defaults to 1.
- **period** (*int*) – The number of entries in the time series data that corresponds to one period of a cyclic signal. For instance, if data is known to possess a weekly seasonal signal, and if the data is daily data, period should be 7. For daily data with a yearly seasonal signal, period should be 365. Defaults to -1, which uses the statsmodels libarary's freq_to_period function. https://github.com/statsmodels/statsmodels/blob/main/statsmodels/tsa/tsatools.py
- **random_seed** (*int*) – Seed for the random number generator. Defaults to 0.

**Attributes**

| | |
|---|---|
| **hyper-parameter_ranges** | { "degree": Integer(1, 3)} |
| **invalid_frequencies** | [] |
| **modifies_features** | False |
| **modifies_target** | True |
| **name** | Polynomial Decomposer |
| **needs_fitting** | True |
| **training_only** | False |

**Methods**

| | |
|---|---|
| *clone* | Constructs a new component with the same parameters and random state. |
| *default_parameters* | Returns the default parameters for this component. |
| *describe* | Describe a component and its parameters. |
| *determine_periodicity* | Function that uses autocorrelative methods to determine the likely most signficant period of the seasonal signal. |
| *fit* | Fits the PolynomialDecomposer and determine the seasonal signal. |
| *fit_transform* | Removes fitted trend and seasonality from target variable. |
| *get_trend_dataframe* | Return a list of dataframes with 4 columns: signal, trend, seasonality, residual. |
| *inverse_transform* | Adds back fitted trend and seasonality to target variable. |
| *is_freq_valid* | Determines if the given string represents a valid frequency for this decomposer. |
| *load* | Loads component at file path. |
| *parameters* | Returns the parameters which were used to initialize the component. |
| *plot_decomposition* | Plots the decomposition of the target signal. |
| *save* | Saves component at file path. |
| *set_period* | Function to set the component's seasonal period based on the target's seasonality. |
| *transform* | Transforms the target data by removing the polynomial trend and rolling average seasonality. |
| *update_parameters* | Updates the parameter dictionary of the component. |

**clone**(*self*)

Constructs a new component with the same parameters and random state.

**Returns** A new instance of this component with identical parameters and random state.

**default_parameters**(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

**Returns** Default parameters for this component.

**Return type** dict

**describe**(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

**Parameters**

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description as dictionary in the format {"name": name, "parameters": parameters}

**Returns** Returns dictionary if return_dict is True, else None.

**Return type** None or dict

**classmethod determine_periodicity**(*cls*, *X: pandas.DataFrame*, *y: pandas.Series*, *acf_threshold: float = 0.01*, *rel_max_order: int = 5*)

Function that uses autocorrelative methods to determine the likely most signficant period of the seasonal signal.

**Parameters**

- **X** (*pandas.DataFrame*) – The feature data of the time series problem.
- **y** (*pandas.Series*) – The target data of a time series problem.
- **acf_threshold** (*float*) – The threshold for the autocorrelation function to determine the period. Any values below the threshold are considered to be 0 and will not be considered for the period. Defaults to 0.01.
- **rel_max_order** (*int*) – The order of the relative maximum to determine the period. Defaults to 5.

**Returns**

**The integer number of entries in time series data over which the seasonal part of the target data** repeats. If the time series data is in days, then this is the number of days that it takes the target's seasonal signal to repeat. Note: the target data can contain multiple seasonal signals. This function will only return the stronger. E.g. if the target has both weekly and yearly seasonality, the function may return either "7" or "365", depending on which seasonality is more strongly autocorrelated. If no period is detected, returns None.

**Return type** int

**fit**(*self*, *X: pandas.DataFrame*, *y: pandas.Series = None*) → *PolynomialDecomposer*

Fits the PolynomialDecomposer and determine the seasonal signal.

Currently only fits the polynomial detrender. The seasonality is determined by removing the trend from the signal and using statsmodels' seasonal_decompose(). Both the trend and seasonality are currently assumed to be additive.

**Parameters**

- **X** (*pd.DataFrame, optional*) – Conditionally used to build datetime index.
- **y** (*pd.Series*) – Target variable to detrend and deseasonalize.

**Returns** self

**Raises**

- **NotImplementedError** – If the input data has a frequency of "month-begin". This isn't supported by statsmodels decompose as the freqstr "MS" is misinterpreted as milliseconds.
- **ValueError** – If y is None.
- **ValueError** – If target data doesn't have DatetimeIndex AND no Datetime features in features data

**fit_transform**(*self*, *X: pandas.DataFrame*, *y: pandas.Series = None*) → tuple[pandas.DataFrame, pandas.Series]

Removes fitted trend and seasonality from target variable.

**Parameters**

- **X** (*pd.DataFrame, optional*) – Ignored.
- **y** (*pd.Series*) – Target variable to detrend and deseasonalize.

**Returns**

**The first element are the input features returned without modification.** The second element is the target variable y with the fitted trend removed.

**Return type** tuple of pd.DataFrame, pd.Series

`get_trend_dataframe`(*self*, *X: pandas.DataFrame*, *y: pandas.Series*) → list[pandas.DataFrame]

Return a list of dataframes with 4 columns: signal, trend, seasonality, residual.

Scikit-learn's PolynomialForecaster is used to generate the trend portion of the target data. statsmodel's seasonal_decompose is used to generate the seasonality of the data.

**Parameters**

- **X** (*pd.DataFrame*) – Input data with time series data in index.
- **y** (*pd.Series or pd.DataFrame*) – Target variable data provided as a Series for univariate problems or a DataFrame for multivariate problems.

**Returns**

**Each DataFrame contains the columns "signal", "trend", "seasonality" and "residual,"** with the latter 3 column values being the decomposed elements of the target data. The "signal" column is simply the input target signal but reindexed with a datetime index to match the input features.

**Return type** list of pd.DataFrame

**Raises**

- **TypeError** – If X does not have time-series data in the index.
- **ValueError** – If time series index of X does not have an inferred frequency.
- **ValueError** – If the forecaster associated with the detrender has not been fit yet.
- **TypeError** – If y is not provided as a pandas Series or DataFrame.

`inverse_transform`(*self*, *y_t: pandas.Series*) → tuple[pandas.DataFrame, pandas.Series]

Adds back fitted trend and seasonality to target variable.

The polynomial trend is added back into the signal, calling the detrender's inverse_transform(). Then, the seasonality is projected forward to and added back into the signal.

**Parameters** **y_t** (*pd.Series*) – Target variable.

**Returns**

**The first element are the input features returned without modification.** The second element is the target variable y with the trend and seasonality added back in.

**Return type** tuple of pd.DataFrame, pd.Series

**Raises** **ValueError** – If y is None.

`classmethod is_freq_valid`(*cls*, *freq: str*)

Determines if the given string represents a valid frequency for this decomposer.

**Parameters** **freq** (*str*) – A frequency to validate. See the pandas docs at https://pandas.pydata.org/pandas-docs/stable/user_guide/timeseries.html#offset-aliases for options.

**Returns** boolean representing whether the frequency is valid or not.

`static load`(*file_path*)

Loads component at file path.

**Parameters** **file_path** (*str*) – Location to load file.

**Returns** ComponentBase object

**property parameters**(*self*)

Returns the parameters which were used to initialize the component.

**plot_decomposition**(*self*, *X: pandas.DataFrame*, *y: pandas.Series*, *show: bool = False*) → tuple[matplotlib.pyplot.Figure, list]

Plots the decomposition of the target signal.

**Parameters**

- **X** (*pd.DataFrame*) – Input data with time series data in index.
- **y** (*pd.Series or pd.DataFrame*) – Target variable data provided as a Series for univariate problems or a DataFrame for multivariate problems.
- **show** (*bool*) – Whether to display the plot or not. Defaults to False.

**Returns**

**The figure and axes that have the decompositions** plotted on them

**Return type** matplotlib.pyplot.Figure, list[matplotlib.pyplot.Axes]

**save**(*self*, *file_path*, *pickle_protocol=cloudpickle.DEFAULT_PROTOCOL*)

Saves component at file path.

**Parameters**

- **file_path** (*str*) – Location to save file.
- **pickle_protocol** (*int*) – The pickle data stream format.

**set_period**(*self*, *X: pandas.DataFrame*, *y: pandas.Series*, *acf_threshold: float = 0.01*, *rel_max_order: int = 5*)

Function to set the component's seasonal period based on the target's seasonality.

**Parameters**

- **X** (*pandas.DataFrame*) – The feature data of the time series problem.
- **y** (*pandas.Series*) – The target data of a time series problem.
- **acf_threshold** (*float*) – The threshold for the autocorrelation function to determine the period. Any values below the threshold are considered to be 0 and will not be considered for the period. Defaults to 0.01.
- **rel_max_order** (*int*) – The order of the relative maximum to determine the period. Defaults to 5.

**transform**(*self*, *X: pandas.DataFrame*, *y: pandas.Series = None*) → tuple[pandas.DataFrame, pandas.Series]

Transforms the target data by removing the polynomial trend and rolling average seasonality.

Applies the fit polynomial detrender to the target data, removing the additive polynomial trend. Then, utilizes the first period's worth of seasonal data determined in the .fit() function to extrapolate the seasonal signal of the data to be transformed. This seasonal signal is also assumed to be additive and is removed.

**Parameters**

- **X** (*pd.DataFrame, optional*) – Conditionally used to build datetime index.
- **y** (*pd.Series*) – Target variable to detrend and deseasonalize.

**Returns**

**The input features are returned without modification. The target** variable y is detrended and deseasonalized.

**Return type** tuple of pd.DataFrame, pd.Series

**Raises** `ValueError` – If target data doesn't have DatetimeIndex AND no Datetime features in features data

`update_parameters`(*self*, *update_dict*, *reset_fit=True*)

Updates the parameter dictionary of the component.

**Parameters**

- `update_dict` (*dict*) – A dict of parameters to update.
- `reset_fit` (*bool, optional*) – If True, will set *_is_fitted* to False.

`class evalml.pipelines.components.ProphetRegressor`(*time_index: Optional[Hashable] = None*, *changepoint_prior_scale: float = 0.05*, *seasonality_prior_scale: int = 10*, *holidays_prior_scale: int = 10*, *seasonality_mode: str = 'additive'*, *stan_backend: str = 'CMDSTANPY'*, *interval_width: float = 0.95*, *random_seed: Union[int, float] = 0*, ***kwargs*)

Prophet is a procedure for forecasting time series data based on an additive model where non-linear trends are fit with yearly, weekly, and daily seasonality, plus holiday effects. It works best with time series that have strong seasonal effects and several seasons of historical data. Prophet is robust to missing data and shifts in the trend, and typically handles outliers well.

More information here: https://facebook.github.io/prophet/

**Parameters**

- `time_index` (*str*) – Specifies the name of the column in X that provides the datetime objects. Defaults to None.
- `changepoint_prior_scale` (*float*) – Determines the strength of the sparse prior for fitting on rate changes. Increasing this value increases the flexibility of the trend. Defaults to 0.05.
- `seasonality_prior_scale` (*int*) – Similar to changepoint_prior_scale. Adjusts the extent to which the seasonality model will fit the data. Defaults to 10.
- `holidays_prior_scale` (*int*) – Similar to changepoint_prior_scale. Adjusts the extent to which holidays will fit the data. Defaults to 10.
- `seasonality_mode` (*str*) – Determines how this component fits the seasonality. Options are "additive" and "multiplicative". Defaults to "additive".
- `stan_backend` (*str*) – Determines the backend that should be used to run Prophet. Options are "CMDSTANPY" and "PYSTAN". Defaults to "CMDSTANPY".
- `interval_width` (*float*) – Determines the confidence of the prediction interval range when calling *get_prediction_intervals*. Accepts values in the range (0,1). Defaults to 0.95.
- `random_seed` (*int*) – Seed for the random number generator. Defaults to 0.

**Attributes**

| | |
|---|---|
| **hyper-parameter_ranges** | { "changepoint_prior_scale": Real(0.001, 0.5), "seasonality_prior_scale": Real(0.01, 10), "holidays_prior_scale": Real(0.01, 10), "seasonality_mode": ["additive", "multiplicative"],} |
| **model_family** | ModelFamily.PROPHET |
| **modifies_features** | True |
| **modifies_target** | False |
| **name** | Prophet Regressor |
| **supported_problem_types** | [ProblemTypes.TIME_SERIES_REGRESSION] |
| **training_only** | False |

**Methods**

| | |
|---|---|
| *build_prophet_df* | Build the Prophet data to pass fit and predict on. |
| *clone* | Constructs a new component with the same parameters and random state. |
| *default_parameters* | Returns the default parameters for this component. |
| *describe* | Describe a component and its parameters. |
| *feature_importance* | Returns array of 0's with len(1) as feature_importance is not defined for Prophet regressor. |
| *fit* | Fits Prophet regressor component to data. |
| *get_params* | Get parameters for the Prophet regressor. |
| *get_prediction_intervals* | Find the prediction intervals using the fitted ProphetRegressor. |
| *load* | Loads component at file path. |
| *needs_fitting* | Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances. |
| *parameters* | Returns the parameters which were used to initialize the component. |
| *predict* | Make predictions using fitted Prophet regressor. |
| *predict_proba* | Make probability estimates for labels. |
| *save* | Saves component at file path. |
| *update_parameters* | Updates the parameter dictionary of the component. |

**static build_prophet_df**(*X: pandas.DataFrame*, *y: Optional[pandas.Series] = None*, *time_index: str = 'ds'*) → pandas.DataFrame

Build the Prophet data to pass fit and predict on.

**clone**(*self*)

Constructs a new component with the same parameters and random state.

**Returns** A new instance of this component with identical parameters and random state.

**default_parameters**(*cls*) → dict

Returns the default parameters for this component.

**Returns** Default parameters for this component.

**Return type** dict

**describe**(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

**Parameters**

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description as dictionary in the format {"name": name, "parameters": parameters}

**Returns** Returns dictionary if return_dict is True, else None.

**Return type** None or dict

**property feature_importance**(*self*) → numpy.ndarray

Returns array of 0's with len(1) as feature_importance is not defined for Prophet regressor.

**fit**(*self*, *X: pandas.DataFrame*, *y: Optional[pandas.Series] = None*)

Fits Prophet regressor component to data.

**Parameters**

- **X** (*pd.DataFrame*) – The input training data of shape [n_samples, n_features].
- **y** (*pd.Series*) – The target training data of length [n_samples].

**Returns** self

**get_params**(*self*) → dict

Get parameters for the Prophet regressor.

**get_prediction_intervals**(*self*, *X: pandas.DataFrame*, *y: Optional[pandas.Series] = None*, *coverage: List[float] = None*, *predictions: pandas.Series = None*) → Dict[str, pandas.Series]

Find the prediction intervals using the fitted ProphetRegressor.

**Parameters**

- **X** (*pd.DataFrame*) – Data of shape [n_samples, n_features].
- **y** (*pd.Series*) – Target data. Ignored.
- **coverage** (*List[float]*) – A list of floats between the values 0 and 1 that the upper and lower bounds of the prediction interval should be calculated for.
- **predictions** (*pd.Series*) – Not used for Prophet estimator.

**Returns** Prediction intervals, keys are in the format {coverage}_lower or {coverage}_upper.

**Return type** dict

**static load**(*file_path*)

Loads component at file path.

**Parameters** **file_path** (*str*) – Location to load file.

**Returns** ComponentBase object

**needs_fitting**(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need to be fit or whose fit methods do nothing.

**Returns** True.

**property parameters**(*self*)

Returns the parameters which were used to initialize the component.

**predict**(*self*, *X: pandas.DataFrame*, *y: Optional[pandas.Series] = None*) → pandas.Series

Make predictions using fitted Prophet regressor.

**Parameters**

- **X** (*pd.DataFrame*) – Data of shape [n_samples, n_features].
- **y** (*pd.Series*) – Target data. Ignored.

**Returns** Predicted values.

**Return type** pd.Series

**predict_proba**(*self*, *X: pandas.DataFrame*) → pandas.Series

Make probability estimates for labels.

**Parameters** **X** (*pd.DataFrame*) – Features.

**Returns** Probability estimates.

**Return type** pd.Series

**Raises** **MethodPropertyNotFoundError** – If estimator does not have a predict_proba method or a component_obj that implements predict_proba.

**save**(*self*, *file_path*, *pickle_protocol=cloudpickle.DEFAULT_PROTOCOL*)

Saves component at file path.

**Parameters**

- **file_path** (*str*) – Location to save file.
- **pickle_protocol** (*int*) – The pickle data stream format.

**update_parameters**(*self*, *update_dict*, *reset_fit=True*)

Updates the parameter dictionary of the component.

**Parameters**

- **update_dict** (*dict*) – A dict of parameters to update.
- **reset_fit** (*bool, optional*) – If True, will set *_is_fitted* to False.

**class** evalml.pipelines.components.**RandomForestClassifier**(*n_estimators=100*, *max_depth=6*, *n_jobs=- 1*, *random_seed=0*, ***kwargs*)

Random Forest Classifier.

**Parameters**

- **n_estimators** (*float*) – The number of trees in the forest. Defaults to 100.
- **max_depth** (*int*) – Maximum tree depth for base learners. Defaults to 6.
- **n_jobs** (*int or None*) – Number of jobs to run in parallel. -1 uses all processes. Defaults to -1.
- **random_seed** (*int*) – Seed for the random number generator. Defaults to 0.

**Attributes**

| | |
|---|---|
| **hyper-parameter_ranges** | { "n_estimators": Integer(10, 1000), "max_depth": Integer(1, 10),} |
| **model_family** | ModelFamily.RANDOM_FOREST |
| **modifies_features** | True |
| **modifies_target** | False |
| **name** | Random Forest Classifier |
| **supported_problem_types** | [ ProblemTypes.BINARY, ProblemTypes.MULTICLASS, ProblemTypes.TIME_SERIES_BINARY, ProblemTypes.TIME_SERIES_MULTICLASS,] |
| **training_only** | False |

**Methods**

| | |
|---|---|
| *clone* | Constructs a new component with the same parameters and random state. |
| *default_parameters* | Returns the default parameters for this component. |
| *describe* | Describe a component and its parameters. |
| *feature_importance* | Returns importance associated with each feature. |
| *fit* | Fits estimator to data. |
| *get_prediction_intervals* | Find the prediction intervals using the fitted regressor. |
| *load* | Loads component at file path. |
| *needs_fitting* | Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances. |
| *parameters* | Returns the parameters which were used to initialize the component. |
| *predict* | Make predictions using selected features. |
| *predict_proba* | Make probability estimates for labels. |
| *save* | Saves component at file path. |
| *update_parameters* | Updates the parameter dictionary of the component. |

**clone**(*self*)

Constructs a new component with the same parameters and random state.

**Returns** A new instance of this component with identical parameters and random state.

**default_parameters**(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

**Returns** Default parameters for this component.

**Return type** dict

**describe**(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

**Parameters**

- **print_name** (*bool, optional*) – whether to print name of component

- **return_dict** (*bool, optional*) – whether to return description as dictionary in the format {"name": name, "parameters": parameters}

**Returns** Returns dictionary if return_dict is True, else None.

**Return type** None or dict

**property feature_importance**(*self*) → pandas.Series

Returns importance associated with each feature.

**Returns** Importance associated with each feature.

**Return type** np.ndarray

**Raises** **MethodPropertyNotFoundError** – If estimator does not have a feature_importance method or a component_obj that implements feature_importance.

**fit**(*self, X: pandas.DataFrame, y: Optional[pandas.Series] = None*)

Fits estimator to data.

**Parameters**

- **X** (*pd.DataFrame*) – The input training data of shape [n_samples, n_features].
- **y** (*pd.Series, optional*) – The target training data of length [n_samples].

**Returns** self

**get_prediction_intervals**(*self, X: pandas.DataFrame, y: Optional[pandas.Series] = None, coverage: List[float] = None, predictions: pandas.Series = None*) → Dict[str, pandas.Series]

Find the prediction intervals using the fitted regressor.

This function takes the predictions of the fitted estimator and calculates the rolling standard deviation across all predictions using a window size of 5. The lower and upper predictions are determined by taking the percent point (quantile) function of the lower tail probability at each bound multiplied by the rolling standard deviation.

**Parameters**

- **X** (*pd.DataFrame*) – Data of shape [n_samples, n_features].
- **y** (*pd.Series*) – Target data. Ignored.
- **coverage** (*list[float]*) – A list of floats between the values 0 and 1 that the upper and lower bounds of the prediction interval should be calculated for.
- **predictions** (*pd.Series*) – Optional list of predictions to use. If None, will generate predictions using *X*.

**Returns** Prediction intervals, keys are in the format {coverage}_lower or {coverage}_upper.

**Return type** dict

**Raises** **MethodPropertyNotFoundError** – If the estimator does not support Time Series Regression as a problem type.

**static load**(*file_path*)

Loads component at file path.

**Parameters** **file_path** (*str*) – Location to load file.

**Returns** ComponentBase object

**needs_fitting**(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need to be fit or whose fit methods do nothing.

**Returns** True.

**property parameters**(*self*)

Returns the parameters which were used to initialize the component.

**predict**(*self*, *X: pandas.DataFrame*) → pandas.Series

Make predictions using selected features.

**Parameters** **X** (*pd.DataFrame*) – Data of shape [n_samples, n_features].

**Returns** Predicted values.

**Return type** pd.Series

**Raises** **MethodPropertyNotFoundError** – If estimator does not have a predict method or a component_obj that implements predict.

**predict_proba**(*self*, *X: pandas.DataFrame*) → pandas.Series

Make probability estimates for labels.

**Parameters** **X** (*pd.DataFrame*) – Features.

**Returns** Probability estimates.

**Return type** pd.Series

**Raises** **MethodPropertyNotFoundError** – If estimator does not have a predict_proba method or a component_obj that implements predict_proba.

**save**(*self*, *file_path*, *pickle_protocol=cloudpickle.DEFAULT_PROTOCOL*)

Saves component at file path.

**Parameters**

- **file_path** (*str*) – Location to save file.
- **pickle_protocol** (*int*) – The pickle data stream format.

**update_parameters**(*self*, *update_dict*, *reset_fit=True*)

Updates the parameter dictionary of the component.

**Parameters**

- **update_dict** (*dict*) – A dict of parameters to update.
- **reset_fit** (*bool, optional*) – If True, will set *_is_fitted* to False.

**class** evalml.pipelines.components.**RandomForestRegressor**(*n_estimators: int = 100*, *max_depth: int = 6*, *n_jobs: int = - 1*, *random_seed: Union[int, float] = 0*, ***kwargs*)

Random Forest Regressor.

**Parameters**

- **n_estimators** (*float*) – The number of trees in the forest. Defaults to 100.
- **max_depth** (*int*) – Maximum tree depth for base learners. Defaults to 6.

- **n_jobs** (*int or None*) – Number of jobs to run in parallel. -1 uses all processes. Defaults to -1.
- **random_seed** (*int*) – Seed for the random number generator. Defaults to 0.

**Attributes**

| | |
|---|---|
| **hyper-parameter_ranges** | { "n_estimators": Integer(10, 1000), "max_depth": Integer(1, 32),} |
| **model_family** | ModelFamily.RANDOM_FOREST |
| **modifies_features** | True |
| **modifies_target** | False |
| **name** | Random Forest Regressor |
| **supported_problem_types** | [ ProblemTypes.REGRESSION, ProblemTypes.TIME_SERIES_REGRESSION,] |
| **training_only** | False |

**Methods**

| | |
|---|---|
| *clone* | Constructs a new component with the same parameters and random state. |
| *default_parameters* | Returns the default parameters for this component. |
| *describe* | Describe a component and its parameters. |
| *feature_importance* | Returns importance associated with each feature. |
| *fit* | Fits estimator to data. |
| *get_prediction_intervals* | Find the prediction intervals using the fitted RandomForestRegressor. |
| *load* | Loads component at file path. |
| *needs_fitting* | Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances. |
| *parameters* | Returns the parameters which were used to initialize the component. |
| *predict* | Make predictions using selected features. |
| *predict_proba* | Make probability estimates for labels. |
| *save* | Saves component at file path. |
| *update_parameters* | Updates the parameter dictionary of the component. |

**clone**(*self*)

Constructs a new component with the same parameters and random state.

**Returns** A new instance of this component with identical parameters and random state.

**default_parameters**(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

**Returns** Default parameters for this component.

**Return type** dict

**describe**(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

**Parameters**

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description as dictionary in the format {"name": name, "parameters": parameters}

**Returns** Returns dictionary if return_dict is True, else None.

**Return type** None or dict

**property feature_importance**(*self*) → pandas.Series

Returns importance associated with each feature.

**Returns** Importance associated with each feature.

**Return type** np.ndarray

**Raises** **MethodPropertyNotFoundError** – If estimator does not have a feature_importance method or a component_obj that implements feature_importance.

**fit**(*self*, *X: pandas.DataFrame*, *y: Optional[pandas.Series] = None*)

Fits estimator to data.

**Parameters**

- **X** (*pd.DataFrame*) – The input training data of shape [n_samples, n_features].
- **y** (*pd.Series, optional*) – The target training data of length [n_samples].

**Returns** self

**get_prediction_intervals**(*self*, *X: pandas.DataFrame*, *y: Optional[pandas.Series] = None*, *coverage: List[float] = None*, *predictions: pandas.Series = None*) → Dict[str, pandas.Series]

Find the prediction intervals using the fitted RandomForestRegressor.

**Parameters**

- **X** (*pd.DataFrame*) – Data of shape [n_samples, n_features].
- **y** (*pd.Series*) – Target data. Optional.
- **coverage** (*list[float]*) – A list of floats between the values 0 and 1 that the upper and lower bounds of the prediction interval should be calculated for.
- **predictions** (*pd.Series*) – Optional list of predictions to use. If None, will generate predictions using *X*.

**Returns** Prediction intervals, keys are in the format {coverage}_lower or {coverage}_upper.

**Return type** dict

**static load**(*file_path*)

Loads component at file path.

**Parameters** **file_path** (*str*) – Location to load file.

**Returns** ComponentBase object

**needs_fitting**(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need to be fit or whose fit methods do nothing.

**Returns** True.

**property parameters**(*self*)

Returns the parameters which were used to initialize the component.

**predict**(*self*, *X: pandas.DataFrame*) → pandas.Series

Make predictions using selected features.

**Parameters** **X** (*pd.DataFrame*) – Data of shape [n_samples, n_features].

**Returns** Predicted values.

**Return type** pd.Series

**Raises** **MethodPropertyNotFoundError** – If estimator does not have a predict method or a component_obj that implements predict.

**predict_proba**(*self*, *X: pandas.DataFrame*) → pandas.Series

Make probability estimates for labels.

**Parameters** **X** (*pd.DataFrame*) – Features.

**Returns** Probability estimates.

**Return type** pd.Series

**Raises** **MethodPropertyNotFoundError** – If estimator does not have a predict_proba method or a component_obj that implements predict_proba.

**save**(*self*, *file_path*, *pickle_protocol=cloudpickle.DEFAULT_PROTOCOL*)

Saves component at file path.

**Parameters**

- **file_path** (*str*) – Location to save file.
- **pickle_protocol** (*int*) – The pickle data stream format.

**update_parameters**(*self*, *update_dict*, *reset_fit=True*)

Updates the parameter dictionary of the component.

**Parameters**

- **update_dict** (*dict*) – A dict of parameters to update.
- **reset_fit** (*bool, optional*) – If True, will set *_is_fitted* to False.

**class** evalml.pipelines.components.**ReplaceNullableTypes**(*random_seed=0*, ***kwargs*)

Transformer to replace features with the new nullable dtypes with a dtype that is compatible in EvalML.

**Attributes**

| | |
|---|---|
| **hyper-parameter_ranges** | None |
| **modifies_features** | True |
| **modifies_target** | {} |
| **name** | Replace Nullable Types Transformer |
| **training_only** | False |

**Methods**

| | |
|---|---|
| *clone* | Constructs a new component with the same parameters and random state. |
| *default_parameters* | Returns the default parameters for this component. |
| *describe* | Describe a component and its parameters. |
| *fit* | Fits component to data. |
| *fit_transform* | Substitutes non-nullable types for the new pandas nullable types in the data and target data. |
| *load* | Loads component at file path. |
| *needs_fitting* | Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances. |
| *parameters* | Returns the parameters which were used to initialize the component. |
| *save* | Saves component at file path. |
| *transform* | Transforms data by replacing columns that contain nullable types with the appropriate replacement type. |
| *update_parameters* | Updates the parameter dictionary of the component. |

**clone**(*self*)

Constructs a new component with the same parameters and random state.

**Returns** A new instance of this component with identical parameters and random state.

**default_parameters**(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

**Returns** Default parameters for this component.

**Return type** dict

**describe**(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

**Parameters**

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description as dictionary in the format {"name": name, "parameters": parameters}

**Returns** Returns dictionary if return_dict is True, else None.

**Return type** None or dict

**fit**(*self*, *X*, *y=None*)

Fits component to data.

**Parameters**

- **X** (*pd.DataFrame*) – The input training data of shape [n_samples, n_features].
- **y** (*pd.Series, optional*) – The target training data of length [n_samples].

**Returns** self

**fit_transform**(*self*, *X*, *y=None*)

Substitutes non-nullable types for the new pandas nullable types in the data and target data.

**Parameters**

- **X** (*pd.DataFrame, optional*) – Input features.
- **y** (*pd.Series*) – Target data.

**Returns** The input features and target data with the non-nullable types set.

**Return type** tuple of pd.DataFrame, pd.Series

**static load**(*file_path*)

Loads component at file path.

**Parameters** **file_path** (*str*) – Location to load file.

**Returns** ComponentBase object

**needs_fitting**(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need to be fit or whose fit methods do nothing.

**Returns** True.

**property parameters**(*self*)

Returns the parameters which were used to initialize the component.

**save**(*self*, *file_path*, *pickle_protocol=cloudpickle.DEFAULT_PROTOCOL*)

Saves component at file path.

**Parameters**

- **file_path** (*str*) – Location to save file.
- **pickle_protocol** (*int*) – The pickle data stream format.

**transform**(*self*, *X*, *y=None*)

Transforms data by replacing columns that contain nullable types with the appropriate replacement type.

"float64" for nullable integers and "category" for nullable booleans.

**Parameters**

- **X** (*pd.DataFrame*) – Data to transform
- **y** (*pd.Series, optional*) – Target data to transform

**Returns** Transformed X pd.Series: Transformed y

**Return type** pd.DataFrame

**update_parameters**(*self*, *update_dict*, *reset_fit=True*)

Updates the parameter dictionary of the component.

**Parameters**

- **update_dict** (*dict*) – A dict of parameters to update.
- **reset_fit** (*bool, optional*) – If True, will set *_is_fitted* to False.

**class** evalml.pipelines.components.**RFClassifierRFESelector**(*step=0.2*, *min_features_to_select=1*, *cv=None*, *scoring=None*, *n_jobs=- 1*, *n_estimators=10*, *max_depth=None*, *random_seed=0*, ***kwargs*)

Selects relevant features using recursive feature elimination with a Random Forest Classifier.

**Parameters**

- **step** (*int, float*) – The number of features to eliminate in each iteration. If an integer is specified this will represent the number of features to eliminate. If a float is specified this represents the percentage of features to eliminate each iteration. The last iteration may drop fewer than this number of features in order to satisfy the min_features_to_select constraint. Defaults to 0.2.
- **min_features_to_select** (*int*) – The minimum number of features to return. Defaults to 1.
- **cv** (*int or None*) – Number of folds to use for the cross-validation splitting strategy. Defaults to None which will use 5 folds.
- **scoring** (*str, callable or None*) – A string or scorer callable object to specify the scoring method.
- **n_jobs** (*int or None*) – Number of jobs to run in parallel. -1 uses all processes. Defaults to -1.
- **n_estimators** (*int*) – The number of trees in the forest. Defaults to 10.
- **max_depth** (*int*) – Maximum tree depth for base learners. Defaults to None.
- **random_seed** (*int*) – Seed for the random number generator. Defaults to 0.

**Attributes**

| | |
|---|---|
| **hyper-parameter_ranges** | { "step": Real(0.05, 0.25)} |
| **modifies_features** | True |
| **modifies_target** | False |
| **name** | RFE Selector with RF Classifier |
| **training_only** | False |

**Methods**

| | |
|---|---|
| *clone* | Constructs a new component with the same parameters and random state. |
| *default_parameters* | Returns the default parameters for this component. |
| *describe* | Describe a component and its parameters. |
| *fit* | Fits component to data. |
| *fit_transform* | Fit and transform data using the feature selector. |
| *get_names* | Get names of selected features. |
| *load* | Loads component at file path. |
| *needs_fitting* | Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances. |
| *parameters* | Returns the parameters which were used to initialize the component. |
| *save* | Saves component at file path. |
| *transform* | Transforms input data by selecting features. If the component_obj does not have a transform method, will raise an MethodPropertyNotFoundError exception. |
| *update_parameters* | Updates the parameter dictionary of the component. |

**clone**(*self*)

Constructs a new component with the same parameters and random state.

**Returns** A new instance of this component with identical parameters and random state.

**default_parameters**(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

**Returns** Default parameters for this component.

**Return type** dict

**describe**(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

**Parameters**

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description as dictionary in the format {"name": name, "parameters": parameters}

**Returns** Returns dictionary if return_dict is True, else None.

**Return type** None or dict

**fit**(*self*, *X*, *y=None*)

Fits component to data.

**Parameters**

- **X** (*pd.DataFrame*) – The input training data of shape [n_samples, n_features]
- **y** (*pd.Series, optional*) – The target training data of length [n_samples]

**Returns** self

**Raises** `MethodPropertyNotFoundError` – If component does not have a fit method or a component_obj that implements fit.

`fit_transform`(*self*, *X*, *y=None*)

Fit and transform data using the feature selector.

**Parameters**

- `X` (*pd.DataFrame*) – The input training data of shape [n_samples, n_features].
- `y` (*pd.Series, optional*) – The target training data of length [n_samples].

**Returns** Transformed data.

**Return type** pd.DataFrame

`get_names`(*self*)

Get names of selected features.

**Returns** List of the names of features selected.

**Return type** list[str]

`static load`(*file_path*)

Loads component at file path.

**Parameters** `file_path` (*str*) – Location to load file.

**Returns** ComponentBase object

`needs_fitting`(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need to be fit or whose fit methods do nothing.

**Returns** True.

`property parameters`(*self*)

Returns the parameters which were used to initialize the component.

`save`(*self*, *file_path*, *pickle_protocol=cloudpickle.DEFAULT_PROTOCOL*)

Saves component at file path.

**Parameters**

- `file_path` (*str*) – Location to save file.
- `pickle_protocol` (*int*) – The pickle data stream format.

`transform`(*self*, *X*, *y=None*)

Transforms input data by selecting features. If the component_obj does not have a transform method, will raise an MethodPropertyNotFoundError exception.

**Parameters**

- `X` (*pd.DataFrame*) – Data to transform.
- `y` (*pd.Series, optional*) – Target data. Ignored.

**Returns** Transformed X

**Return type** pd.DataFrame

**Raises** `MethodPropertyNotFoundError` – If feature selector does not have a transform method or a component_obj that implements transform

**update_parameters**(*self*, *update_dict*, *reset_fit=True*)

Updates the parameter dictionary of the component.

**Parameters**

- **update_dict** (*dict*) – A dict of parameters to update.
- **reset_fit** (*bool, optional*) – If True, will set *_is_fitted* to False.

**class** evalml.pipelines.components.**RFClassifierSelectFromModel**(*number_features=None*, *n_estimators=10*, *max_depth=None*, *percent_features=0.5*, *threshold='median'*, *n_jobs=- 1*, *random_seed=0*, ***kwargs*)

Selects top features based on importance weights using a Random Forest classifier.

**Parameters**

- **number_features** (*int*) – The maximum number of features to select. If both percent_features and number_features are specified, take the greater number of features. Defaults to None.
- **n_estimators** (*int*) – The number of trees in the forest. Defaults to 10.
- **max_depth** (*int*) – Maximum tree depth for base learners. Defaults to None.
- **percent_features** (*float*) – Percentage of features to use. If both percent_features and number_features are specified, take the greater number of features. Defaults to 0.5.
- **threshold** (*string or float*) – The threshold value to use for feature selection. Features whose importance is greater or equal are kept while the others are discarded. If "median", then the threshold value is the median of the feature importances. A scaling factor (e.g., "1.25*mean") may also be used. Defaults to median.
- **n_jobs** (*int or None*) – Number of jobs to run in parallel. -1 uses all processes. Defaults to -1.
- **random_seed** (*int*) – Seed for the random number generator. Defaults to 0.

**Attributes**

| | |
|---|---|
| **hyper-parameter_ranges** | { "percent_features": Real(0.01, 1), "threshold": ["mean", "median"],} |
| **modifies_features** | True |
| **modifies_target** | False |
| **name** | RF Classifier Select From Model |
| **training_only** | False |

**Methods**

| | |
|---|---|
| *clone* | Constructs a new component with the same parameters and random state. |
| *default_parameters* | Returns the default parameters for this component. |
| *describe* | Describe a component and its parameters. |
| *fit* | Fits component to data. |
| *fit_transform* | Fit and transform data using the feature selector. |
| *get_names* | Get names of selected features. |
| *load* | Loads component at file path. |
| *needs_fitting* | Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances. |
| *parameters* | Returns the parameters which were used to initialize the component. |
| *save* | Saves component at file path. |
| *transform* | Transforms input data by selecting features. If the component_obj does not have a transform method, will raise an MethodPropertyNotFoundError exception. |
| *update_parameters* | Updates the parameter dictionary of the component. |

**clone**(*self*)

Constructs a new component with the same parameters and random state.

**Returns** A new instance of this component with identical parameters and random state.

**default_parameters**(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

**Returns** Default parameters for this component.

**Return type** dict

**describe**(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

**Parameters**

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description as dictionary in the format {"name": name, "parameters": parameters}

**Returns** Returns dictionary if return_dict is True, else None.

**Return type** None or dict

**fit**(*self*, *X*, *y=None*)

Fits component to data.

**Parameters**

- **X** (*pd.DataFrame*) – The input training data of shape [n_samples, n_features]
- **y** (*pd.Series, optional*) – The target training data of length [n_samples]

**Returns** self

**Raises** `MethodPropertyNotFoundError` – If component does not have a fit method or a component_obj that implements fit.

`fit_transform`(*self*, *X*, *y=None*)

Fit and transform data using the feature selector.

**Parameters**

- `X` (*pd.DataFrame*) – The input training data of shape [n_samples, n_features].
- `y` (*pd.Series, optional*) – The target training data of length [n_samples].

**Returns** Transformed data.

**Return type** pd.DataFrame

`get_names`(*self*)

Get names of selected features.

**Returns** List of the names of features selected.

**Return type** list[str]

`static load`(*file_path*)

Loads component at file path.

**Parameters** `file_path` (*str*) – Location to load file.

**Returns** ComponentBase object

`needs_fitting`(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need to be fit or whose fit methods do nothing.

**Returns** True.

`property parameters`(*self*)

Returns the parameters which were used to initialize the component.

`save`(*self*, *file_path*, *pickle_protocol=cloudpickle.DEFAULT_PROTOCOL*)

Saves component at file path.

**Parameters**

- `file_path` (*str*) – Location to save file.
- `pickle_protocol` (*int*) – The pickle data stream format.

`transform`(*self*, *X*, *y=None*)

Transforms input data by selecting features. If the component_obj does not have a transform method, will raise an MethodPropertyNotFoundError exception.

**Parameters**

- `X` (*pd.DataFrame*) – Data to transform.
- `y` (*pd.Series, optional*) – Target data. Ignored.

**Returns** Transformed X

**Return type** pd.DataFrame

**Raises** `MethodPropertyNotFoundError` – If feature selector does not have a transform method or a component_obj that implements transform

**update_parameters**(*self*, *update_dict*, *reset_fit=True*)

Updates the parameter dictionary of the component.

**Parameters**

- **update_dict** (*dict*) – A dict of parameters to update.
- **reset_fit** (*bool, optional*) – If True, will set *_is_fitted* to False.

**class** evalml.pipelines.components.**RFRegressorRFESelector**(*step=0.2*, *min_features_to_select=1*, *cv=None*, *scoring=None*, *n_jobs=- 1*, *n_estimators=10*, *max_depth=None*, *random_seed=0*, ***kwargs*)

Selects relevant features using recursive feature elimination with a Random Forest Regressor.

**Parameters**

- **step** (*int, float*) – The number of features to eliminate in each iteration. If an integer is specified this will represent the number of features to eliminate. If a float is specified this represents the percentage of features to eliminate each iteration. The last iteration may drop fewer than this number of features in order to satisfy the min_features_to_select constraint. Defaults to 0.2.
- **min_features_to_select** (*int*) – The minimum number of features to return. Defaults to 1.
- **cv** (*int or None*) – Number of folds to use for the cross-validation splitting strategy. Defaults to None which will use 5 folds.
- **scoring** (*str, callable or None*) – A string or scorer callable object to specify the scoring method.
- **n_jobs** (*int or None*) – Number of jobs to run in parallel. -1 uses all processes. Defaults to -1.
- **n_estimators** (*int*) – The number of trees in the forest. Defaults to 10.
- **max_depth** (*int*) – Maximum tree depth for base learners. Defaults to None.
- **random_seed** (*int*) – Seed for the random number generator. Defaults to 0.

**Attributes**

| | |
|---|---|
| **hyper-parameter_ranges** | { "step": Real(0.05, 0.25)} |
| **modifies_features** | True |
| **modifies_target** | False |
| **name** | RFE Selector with RF Regressor |
| **training_only** | False |

**Methods**

| | |
|---|---|
| *clone* | Constructs a new component with the same parameters and random state. |
| *default_parameters* | Returns the default parameters for this component. |
| *describe* | Describe a component and its parameters. |
| *fit* | Fits component to data. |
| *fit_transform* | Fit and transform data using the feature selector. |
| *get_names* | Get names of selected features. |
| *load* | Loads component at file path. |
| *needs_fitting* | Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances. |
| *parameters* | Returns the parameters which were used to initialize the component. |
| *save* | Saves component at file path. |
| *transform* | Transforms input data by selecting features. If the component_obj does not have a transform method, will raise an MethodPropertyNotFoundError exception. |
| *update_parameters* | Updates the parameter dictionary of the component. |

**clone**(*self*)

Constructs a new component with the same parameters and random state.

**Returns** A new instance of this component with identical parameters and random state.

**default_parameters**(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

**Returns** Default parameters for this component.

**Return type** dict

**describe**(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

**Parameters**

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description as dictionary in the format {"name": name, "parameters": parameters}

**Returns** Returns dictionary if return_dict is True, else None.

**Return type** None or dict

**fit**(*self*, *X*, *y=None*)

Fits component to data.

**Parameters**

- **X** (*pd.DataFrame*) – The input training data of shape [n_samples, n_features]
- **y** (*pd.Series, optional*) – The target training data of length [n_samples]

**Returns** self

**Raises** `MethodPropertyNotFoundError` – If component does not have a fit method or a component_obj that implements fit.

`fit_transform`(*self*, *X*, *y=None*)

Fit and transform data using the feature selector.

**Parameters**

- `X` (*pd.DataFrame*) – The input training data of shape [n_samples, n_features].
- `y` (*pd.Series, optional*) – The target training data of length [n_samples].

**Returns** Transformed data.

**Return type** pd.DataFrame

`get_names`(*self*)

Get names of selected features.

**Returns** List of the names of features selected.

**Return type** list[str]

`static load`(*file_path*)

Loads component at file path.

**Parameters** `file_path` (*str*) – Location to load file.

**Returns** ComponentBase object

`needs_fitting`(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need to be fit or whose fit methods do nothing.

**Returns** True.

`property parameters`(*self*)

Returns the parameters which were used to initialize the component.

`save`(*self*, *file_path*, *pickle_protocol=cloudpickle.DEFAULT_PROTOCOL*)

Saves component at file path.

**Parameters**

- `file_path` (*str*) – Location to save file.
- `pickle_protocol` (*int*) – The pickle data stream format.

`transform`(*self*, *X*, *y=None*)

Transforms input data by selecting features. If the component_obj does not have a transform method, will raise an MethodPropertyNotFoundError exception.

**Parameters**

- `X` (*pd.DataFrame*) – Data to transform.
- `y` (*pd.Series, optional*) – Target data. Ignored.

**Returns** Transformed X

**Return type** pd.DataFrame

**Raises** `MethodPropertyNotFoundError` – If feature selector does not have a transform method or a component_obj that implements transform

**update_parameters**(*self*, *update_dict*, *reset_fit=True*)

Updates the parameter dictionary of the component.

**Parameters**

- **update_dict** (*dict*) – A dict of parameters to update.
- **reset_fit** (*bool, optional*) – If True, will set *_is_fitted* to False.

**class** evalml.pipelines.components.**RFRegressorSelectFromModel**(*number_features=None*, *n_estimators=10*, *max_depth=None*, *percent_features=0.5*, *threshold='median'*, *n_jobs=- 1*, *random_seed=0*, ***kwargs*)

Selects top features based on importance weights using a Random Forest regressor.

**Parameters**

- **number_features** (*int*) – The maximum number of features to select. If both percent_features and number_features are specified, take the greater number of features. Defaults to 0.5.
- **n_estimators** (*int*) – The number of trees in the forest. Defaults to 10.
- **max_depth** (*int*) – Maximum tree depth for base learners. Defaults to None.
- **percent_features** (*float*) – Percentage of features to use. If both percent_features and number_features are specified, take the greater number of features. Defaults to 0.5.
- **threshold** (*string or float*) – The threshold value to use for feature selection. Features whose importance is greater or equal are kept while the others are discarded. If "median", then the threshold value is the median of the feature importances. A scaling factor (e.g., "1.25*mean") may also be used. Defaults to median.
- **n_jobs** (*int or None*) – Number of jobs to run in parallel. -1 uses all processes. Defaults to -1.
- **random_seed** (*int*) – Seed for the random number generator. Defaults to 0.

**Attributes**

| | |
|---|---|
| **hyper-parameter_ranges** | { "percent_features": Real(0.01, 1), "threshold": ["mean", "median"],} |
| **modifies_features** | True |
| **modifies_target** | False |
| **name** | RF Regressor Select From Model |
| **training_only** | False |

**Methods**

| | |
|---|---|
| *clone* | Constructs a new component with the same parameters and random state. |
| *default_parameters* | Returns the default parameters for this component. |
| *describe* | Describe a component and its parameters. |
| *fit* | Fits component to data. |
| *fit_transform* | Fit and transform data using the feature selector. |
| *get_names* | Get names of selected features. |
| *load* | Loads component at file path. |
| *needs_fitting* | Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances. |
| *parameters* | Returns the parameters which were used to initialize the component. |
| *save* | Saves component at file path. |
| *transform* | Transforms input data by selecting features. If the component_obj does not have a transform method, will raise an MethodPropertyNotFoundError exception. |
| *update_parameters* | Updates the parameter dictionary of the component. |

**clone**(*self*)

Constructs a new component with the same parameters and random state.

**Returns** A new instance of this component with identical parameters and random state.

**default_parameters**(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

**Returns** Default parameters for this component.

**Return type** dict

**describe**(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

**Parameters**

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description as dictionary in the format {"name": name, "parameters": parameters}

**Returns** Returns dictionary if return_dict is True, else None.

**Return type** None or dict

**fit**(*self*, *X*, *y=None*)

Fits component to data.

**Parameters**

- **X** (*pd.DataFrame*) – The input training data of shape [n_samples, n_features]
- **y** (*pd.Series, optional*) – The target training data of length [n_samples]

**Returns** self

**Raises** `MethodPropertyNotFoundError` – If component does not have a fit method or a component_obj that implements fit.

`fit_transform`(*self*, *X*, *y=None*)

Fit and transform data using the feature selector.

**Parameters**

- `X` (*pd.DataFrame*) – The input training data of shape [n_samples, n_features].
- `y` (*pd.Series, optional*) – The target training data of length [n_samples].

**Returns** Transformed data.

**Return type** pd.DataFrame

`get_names`(*self*)

Get names of selected features.

**Returns** List of the names of features selected.

**Return type** list[str]

`static load`(*file_path*)

Loads component at file path.

**Parameters** `file_path` (*str*) – Location to load file.

**Returns** ComponentBase object

`needs_fitting`(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need to be fit or whose fit methods do nothing.

**Returns** True.

`property parameters`(*self*)

Returns the parameters which were used to initialize the component.

`save`(*self*, *file_path*, *pickle_protocol=cloudpickle.DEFAULT_PROTOCOL*)

Saves component at file path.

**Parameters**

- `file_path` (*str*) – Location to save file.
- `pickle_protocol` (*int*) – The pickle data stream format.

`transform`(*self*, *X*, *y=None*)

Transforms input data by selecting features. If the component_obj does not have a transform method, will raise an MethodPropertyNotFoundError exception.

**Parameters**

- `X` (*pd.DataFrame*) – Data to transform.
- `y` (*pd.Series, optional*) – Target data. Ignored.

**Returns** Transformed X

**Return type** pd.DataFrame

**Raises** `MethodPropertyNotFoundError` – If feature selector does not have a transform method or a component_obj that implements transform

**update_parameters**(*self*, *update_dict*, *reset_fit=True*)

Updates the parameter dictionary of the component.

**Parameters**

- **update_dict** (*dict*) – A dict of parameters to update.
- **reset_fit** (*bool, optional*) – If True, will set *_is_fitted* to False.

**class** evalml.pipelines.components.**SelectByType**(*column_types=None*, *exclude=False*, *random_seed=0*, ***kwargs*)

Selects columns by specified Woodwork logical type or semantic tag in input data.

**Parameters**

- **column_types** (*string, ww.LogicalType, list(string), list(ww.LogicalType)*) – List of Woodwork types or tags, used to determine which columns to select or exclude.
- **exclude** (*bool*) – If true, exclude the column_types instead of including them. Defaults to False.
- **random_seed** (*int*) – Seed for the random number generator. Defaults to 0.

**Attributes**

| | |
|---|---|
| **hyper-parameter_ranges** | {} |
| **modifies_features** | True |
| **modifies_target** | False |
| **name** | Select Columns By Type Transformer |
| **needs_fitting** | False |
| **training_only** | False |

**Methods**

| | |
|---|---|
| *clone* | Constructs a new component with the same parameters and random state. |
| *default_parameters* | Returns the default parameters for this component. |
| *describe* | Describe a component and its parameters. |
| *fit* | Fits the transformer by checking if column names are present in the dataset. |
| *fit_transform* | Fits on X and transforms X. |
| *load* | Loads component at file path. |
| *parameters* | Returns the parameters which were used to initialize the component. |
| *save* | Saves component at file path. |
| *transform* | Transforms data X by selecting columns. |
| *update_parameters* | Updates the parameter dictionary of the component. |

**clone**(*self*)

Constructs a new component with the same parameters and random state.

**Returns** A new instance of this component with identical parameters and random state.

`default_parameters`(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

**Returns** Default parameters for this component.

**Return type** dict

`describe`(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

**Parameters**

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description as dictionary in the format {"name": name, "parameters": parameters}

**Returns** Returns dictionary if return_dict is True, else None.

**Return type** None or dict

`fit`(*self*, *X*, *y=None*)

Fits the transformer by checking if column names are present in the dataset.

**Parameters**

- **X** (*pd.DataFrame*) – Data to check.
- **y** (*pd.Series, ignored*) – Targets.

**Returns** self

`fit_transform`(*self*, *X*, *y=None*)

Fits on X and transforms X.

**Parameters**

- **X** (*pd.DataFrame*) – Data to fit and transform.
- **y** (*pd.Series*) – Target data.

**Returns** Transformed X.

**Return type** pd.DataFrame

**Raises** **MethodPropertyNotFoundError** – If transformer does not have a transform method or a component_obj that implements transform.

`static load`(*file_path*)

Loads component at file path.

**Parameters** **file_path** (*str*) – Location to load file.

**Returns** ComponentBase object

`property parameters`(*self*)

Returns the parameters which were used to initialize the component.

`save`(*self*, *file_path*, *pickle_protocol=cloudpickle.DEFAULT_PROTOCOL*)

Saves component at file path.

**Parameters**

- **file_path** (*str*) – Location to save file.
- **pickle_protocol** (*int*) – The pickle data stream format.

**transform**(*self*, *X*, *y=None*)

Transforms data X by selecting columns.

**Parameters**

- **X** (*pd.DataFrame*) – Data to transform.
- **y** (*pd.Series, optional*) – Targets.

**Returns** Transformed X.

**Return type** pd.DataFrame

**update_parameters**(*self*, *update_dict*, *reset_fit=True*)

Updates the parameter dictionary of the component.

**Parameters**

- **update_dict** (*dict*) – A dict of parameters to update.
- **reset_fit** (*bool, optional*) – If True, will set *_is_fitted* to False.

**class** evalml.pipelines.components.**SelectColumns**(*columns=None*, *random_seed=0*, ***kwargs*)

Selects specified columns in input data.

**Parameters**

- **columns** (*list(string)*) – List of column names, used to determine which columns to select. If columns are not present, they will not be selected.
- **random_seed** (*int*) – Seed for the random number generator. Defaults to 0.

**Attributes**

| | |
|---|---|
| **hyper-parameter_ranges** | {} |
| **modifies_features** | True |
| **modifies_target** | False |
| **name** | Select Columns Transformer |
| **needs_fitting** | False |
| **training_only** | False |

**Methods**

| | |
|---|---|
| *clone* | Constructs a new component with the same parameters and random state. |
| *default_parameters* | Returns the default parameters for this component. |
| *describe* | Describe a component and its parameters. |
| *fit* | Fits the transformer by checking if column names are present in the dataset. |
| *fit_transform* | Fits on X and transforms X. |
| *load* | Loads component at file path. |
| *parameters* | Returns the parameters which were used to initialize the component. |
| *save* | Saves component at file path. |
| *transform* | Transform data using fitted column selector component. |
| *update_parameters* | Updates the parameter dictionary of the component. |

**clone**(*self*)

Constructs a new component with the same parameters and random state.

**Returns** A new instance of this component with identical parameters and random state.

**default_parameters**(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

**Returns** Default parameters for this component.

**Return type** dict

**describe**(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

**Parameters**

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description as dictionary in the format {"name": name, "parameters": parameters}

**Returns** Returns dictionary if return_dict is True, else None.

**Return type** None or dict

**fit**(*self*, *X*, *y=None*)

Fits the transformer by checking if column names are present in the dataset.

**Parameters**

- **X** (*pd.DataFrame*) – Data to check.
- **y** (*pd.Series, optional*) – Targets.

**Returns** self

**fit_transform**(*self*, *X*, *y=None*)

Fits on X and transforms X.

**Parameters**

- **X** (*pd.DataFrame*) – Data to fit and transform.

- **y** (*pd.Series*) – Target data.

**Returns** Transformed X.

**Return type** pd.DataFrame

**Raises** **MethodPropertyNotFoundError** – If transformer does not have a transform method or a component_obj that implements transform.

**static load**(*file_path*)

Loads component at file path.

**Parameters** **file_path** (*str*) – Location to load file.

**Returns** ComponentBase object

**property parameters**(*self*)

Returns the parameters which were used to initialize the component.

**save**(*self*, *file_path*, *pickle_protocol=cloudpickle.DEFAULT_PROTOCOL*)

Saves component at file path.

**Parameters**

- **file_path** (*str*) – Location to save file.
- **pickle_protocol** (*int*) – The pickle data stream format.

**transform**(*self*, *X*, *y=None*)

Transform data using fitted column selector component.

**Parameters**

- **X** (*pd.DataFrame*) – The input training data of shape [n_samples, n_features].
- **y** (*pd.Series, optional*) – The target training data of length [n_samples].

**Returns** Transformed data.

**Return type** pd.DataFrame

**update_parameters**(*self*, *update_dict*, *reset_fit=True*)

Updates the parameter dictionary of the component.

**Parameters**

- **update_dict** (*dict*) – A dict of parameters to update.
- **reset_fit** (*bool, optional*) – If True, will set *_is_fitted* to False.

**class** evalml.pipelines.components.**SimpleImputer**(*impute_strategy='most_frequent'*, *fill_value=None*, *random_seed=0*, ***kwargs*)

Imputes missing data according to a specified imputation strategy. Natural language columns are ignored.

**Parameters**

- **impute_strategy** (*string*) – Impute strategy to use. Valid values include "mean", "median", "most_frequent", "constant" for numerical data, and "most_frequent", "constant" for object data types.
- **fill_value** (*string*) – When impute_strategy == "constant", fill_value is used to replace missing data. Defaults to 0 when imputing numerical data and "missing_value" for strings or object data types.
- **random_seed** (*int*) – Seed for the random number generator. Defaults to 0.

**Attributes**

| | |
|---|---|
| **hyper-parameter_ranges** | { "impute_strategy": ["mean", "median", "most_frequent"]} |
| **modifies_features** | True |
| **modifies_target** | False |
| **name** | Simple Imputer |
| **training_only** | False |

**Methods**

| | |
|---|---|
| *clone* | Constructs a new component with the same parameters and random state. |
| *default_parameters* | Returns the default parameters for this component. |
| *describe* | Describe a component and its parameters. |
| *fit* | Fits imputer to data. 'None' values are converted to np.nan before imputation and are treated as the same. |
| *fit_transform* | Fits on X and transforms X. |
| *load* | Loads component at file path. |
| *needs_fitting* | Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances. |
| *parameters* | Returns the parameters which were used to initialize the component. |
| *save* | Saves component at file path. |
| *transform* | Transforms input by imputing missing values. 'None' and np.nan values are treated as the same. |
| *update_parameters* | Updates the parameter dictionary of the component. |

**clone**(*self*)

Constructs a new component with the same parameters and random state.

**Returns** A new instance of this component with identical parameters and random state.

**default_parameters**(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

**Returns** Default parameters for this component.

**Return type** dict

**describe**(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

**Parameters**

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description as dictionary in the format {"name": name, "parameters": parameters}

**Returns** Returns dictionary if return_dict is True, else None.

**Return type** None or dict

`fit`(*self*, *X*, *y=None*)

Fits imputer to data. 'None' values are converted to np.nan before imputation and are treated as the same.

**Parameters**

- **X** (*pd.DataFrame or np.ndarray*) – the input training data of shape [n_samples, n_features]
- **y** (*pd.Series, optional*) – the target training data of length [n_samples]

**Returns** self

**Raises** **ValueError** – if the SimpleImputer receives a dataframe with both Boolean and Categorical data.

`fit_transform`(*self*, *X*, *y=None*)

Fits on X and transforms X.

**Parameters**

- **X** (*pd.DataFrame*) – Data to fit and transform
- **y** (*pd.Series, optional*) – Target data.

**Returns** Transformed X

**Return type** pd.DataFrame

`static load`(*file_path*)

Loads component at file path.

**Parameters** **file_path** (*str*) – Location to load file.

**Returns** ComponentBase object

`needs_fitting`(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need to be fit or whose fit methods do nothing.

**Returns** True.

`property parameters`(*self*)

Returns the parameters which were used to initialize the component.

`save`(*self*, *file_path*, *pickle_protocol=cloudpickle.DEFAULT_PROTOCOL*)

Saves component at file path.

**Parameters**

- **file_path** (*str*) – Location to save file.
- **pickle_protocol** (*int*) – The pickle data stream format.

`transform`(*self*, *X*, *y=None*)

Transforms input by imputing missing values. 'None' and np.nan values are treated as the same.

**Parameters**

- **X** (*pd.DataFrame*) – Data to transform.

- **y** (*pd.Series, optional*) – Ignored.

**Returns** Transformed X

**Return type** pd.DataFrame

**update_parameters**(*self*, *update_dict*, *reset_fit=True*)

Updates the parameter dictionary of the component.

**Parameters**

- **update_dict** (*dict*) – A dict of parameters to update.
- **reset_fit** (*bool, optional*) – If True, will set *_is_fitted* to False.

**class** evalml.pipelines.components.**StackedEnsembleBase**(*final_estimator=None*, *n_jobs=- 1*, *random_seed=0*, ***kwargs*)

Stacked Ensemble Base Class.

**Parameters**

- **final_estimator** (*Estimator or subclass*) – The estimator used to combine the base estimators.
- **n_jobs** (*int or None*) – Integer describing level of parallelism used for pipelines. None and 1 are equivalent. If set to -1, all CPUs are used. For n_jobs greater than -1, (n_cpus + 1 + n_jobs) are used. Defaults to -1. - Note: there could be some multi-process errors thrown for values of *n_jobs != 1*. If this is the case, please use *n_jobs = 1*.
- **random_seed** (*int*) – Seed for the random number generator. Defaults to 0.

**Attributes**

| | |
|---|---|
| **model_family** | ModelFamily.ENSEMBLE |
| **modifies_features** | True |
| **modifies_target** | False |
| **training_only** | False |

**Methods**

| | |
|---|---|
| *clone* | Constructs a new component with the same parameters and random state. |
| *default_parameters* | Returns the default parameters for stacked ensemble classes. |
| *describe* | Describe a component and its parameters. |
| *feature_importance* | Not implemented for StackedEnsembleClassifier and StackedEnsembleRegressor. |
| *fit* | Fits estimator to data. |
| *get_prediction_intervals* | Find the prediction intervals using the fitted regressor. |
| *load* | Loads component at file path. |
| *name* | Returns string name of this component. |
| *needs_fitting* | Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances. |
| *parameters* | Returns the parameters which were used to initialize the component. |
| *predict* | Make predictions using selected features. |
| *predict_proba* | Make probability estimates for labels. |
| *save* | Saves component at file path. |
| *supported_problem_types* | Problem types this estimator supports. |
| *update_parameters* | Updates the parameter dictionary of the component. |

**clone**(*self*)

Constructs a new component with the same parameters and random state.

**Returns** A new instance of this component with identical parameters and random state.

**default_parameters**(*cls*)

Returns the default parameters for stacked ensemble classes.

**Returns** default parameters for this component.

**Return type** dict

**describe**(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

**Parameters**

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description as dictionary in the format {"name": name, "parameters": parameters}

**Returns** Returns dictionary if return_dict is True, else None.

**Return type** None or dict

**property feature_importance**(*self*)

Not implemented for StackedEnsembleClassifier and StackedEnsembleRegressor.

**fit**(*self*, *X: pandas.DataFrame*, *y: Optional[pandas.Series] = None*)

Fits estimator to data.

**Parameters**

- **X** (*pd.DataFrame*) – The input training data of shape [n_samples, n_features].

- **y** (*pd.Series, optional*) – The target training data of length [n_samples].

**Returns** self

**get_prediction_intervals**(*self, X: pandas.DataFrame, y: Optional[pandas.Series] = None, coverage: List[float] = None, predictions: pandas.Series = None*) → Dict[str, pandas.Series]

Find the prediction intervals using the fitted regressor.

This function takes the predictions of the fitted estimator and calculates the rolling standard deviation across all predictions using a window size of 5. The lower and upper predictions are determined by taking the percent point (quantile) function of the lower tail probability at each bound multiplied by the rolling standard deviation.

**Parameters**

- **X** (*pd.DataFrame*) – Data of shape [n_samples, n_features].
- **y** (*pd.Series*) – Target data. Ignored.
- **coverage** (*list[float]*) – A list of floats between the values 0 and 1 that the upper and lower bounds of the prediction interval should be calculated for.
- **predictions** (*pd.Series*) – Optional list of predictions to use. If None, will generate predictions using *X*.

**Returns** Prediction intervals, keys are in the format {coverage}_lower or {coverage}_upper.

**Return type** dict

**Raises** **MethodPropertyNotFoundError** – If the estimator does not support Time Series Regression as a problem type.

**static load**(*file_path*)

Loads component at file path.

**Parameters** **file_path** (*str*) – Location to load file.

**Returns** ComponentBase object

**property name**(*cls*)

Returns string name of this component.

**needs_fitting**(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need to be fit or whose fit methods do nothing.

**Returns** True.

**property parameters**(*self*)

Returns the parameters which were used to initialize the component.

**predict**(*self, X: pandas.DataFrame*) → pandas.Series

Make predictions using selected features.

**Parameters** **X** (*pd.DataFrame*) – Data of shape [n_samples, n_features].

**Returns** Predicted values.

**Return type** pd.Series

**Raises** **MethodPropertyNotFoundError** – If estimator does not have a predict method or a component_obj that implements predict.

**predict_proba**(*self*, *X: pandas.DataFrame*) → pandas.Series

Make probability estimates for labels.

**Parameters** **X** (*pd.DataFrame*) – Features.

**Returns** Probability estimates.

**Return type** pd.Series

**Raises** **MethodPropertyNotFoundError** – If estimator does not have a predict_proba method or a component_obj that implements predict_proba.

**save**(*self*, *file_path*, *pickle_protocol=cloudpickle.DEFAULT_PROTOCOL*)

Saves component at file path.

**Parameters**

- **file_path** (*str*) – Location to save file.
- **pickle_protocol** (*int*) – The pickle data stream format.

**property supported_problem_types**(*cls*)

Problem types this estimator supports.

**update_parameters**(*self*, *update_dict*, *reset_fit=True*)

Updates the parameter dictionary of the component.

**Parameters**

- **update_dict** (*dict*) – A dict of parameters to update.
- **reset_fit** (*bool, optional*) – If True, will set *_is_fitted* to False.

**class** evalml.pipelines.components.**StackedEnsembleClassifier**(*final_estimator=None*, *n_jobs=- 1*, *random_seed=0*, ***kwargs*)

Stacked Ensemble Classifier.

**Parameters**

- **final_estimator** (*Estimator or subclass*) – The classifier used to combine the base estimators. If None, uses ElasticNetClassifier.
- **n_jobs** (*int or None*) – Integer describing level of parallelism used for pipelines. None and 1 are equivalent. If set to -1, all CPUs are used. For n_jobs below -1, (n_cpus + 1 + n_jobs) are used. Defaults to -1. - Note: there could be some multi-process errors thrown for values of *n_jobs != 1*. If this is the case, please use *n_jobs = 1*.
- **random_seed** (*int*) – Seed for the random number generator. Defaults to 0.

### Example

```
>>> from evalml.pipelines.component_graph import ComponentGraph
>>> from evalml.pipelines.components.estimators.classifiers.decision_tree_
↪classifier import DecisionTreeClassifier
>>> from evalml.pipelines.components.estimators.classifiers.elasticnet_classifier␣
↪import ElasticNetClassifier
...
>>> component_graph = {
...     "Decision Tree": [DecisionTreeClassifier(random_seed=3), "X", "y"],
...     "Decision Tree B": [DecisionTreeClassifier(random_seed=4), "X", "y"],
```

(continues on next page)

(continued from previous page)

```
...     "Stacked Ensemble": [
...         StackedEnsembleClassifier(n_jobs=1, final_
↪estimator=DecisionTreeClassifier()),
...         "Decision Tree.x",
...         "Decision Tree B.x",
...         "y",
...     ],
... }
...
>>> cg = ComponentGraph(component_graph)
>>> assert cg.default_parameters == {
...     'Decision Tree Classifier': {'criterion': 'gini',
...                                  'max_features': 'auto',
...                                  'max_depth': 6,
...                                  'min_samples_split': 2,
...                                  'min_weight_fraction_leaf': 0.0},
...     'Stacked Ensemble Classifier': {'final_estimator': ElasticNetClassifier,
...                                     'n_jobs': -1}}
```

**Attributes**

| | |
|---|---|
| **hyper-parameter_ranges** | {} |
| **model_family** | ModelFamily.ENSEMBLE |
| **modifies_features** | True |
| **modifies_target** | False |
| **name** | Stacked Ensemble Classifier |
| **supported_problem_types** | [ ProblemTypes.BINARY, ProblemTypes.MULTICLASS, ProblemTypes.TIME_SERIES_BINARY, ProblemTypes.TIME_SERIES_MULTICLASS,] |
| **training_only** | False |

**Methods**

| | |
|---|---|
| *clone* | Constructs a new component with the same parameters and random state. |
| *default_parameters* | Returns the default parameters for stacked ensemble classes. |
| *describe* | Describe a component and its parameters. |
| *feature_importance* | Not implemented for StackedEnsembleClassifier and StackedEnsembleRegressor. |
| *fit* | Fits estimator to data. |
| *get_prediction_intervals* | Find the prediction intervals using the fitted regressor. |
| *load* | Loads component at file path. |
| *needs_fitting* | Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances. |
| *parameters* | Returns the parameters which were used to initialize the component. |
| *predict* | Make predictions using selected features. |
| *predict_proba* | Make probability estimates for labels. |
| *save* | Saves component at file path. |
| *update_parameters* | Updates the parameter dictionary of the component. |

**clone**(*self*)

Constructs a new component with the same parameters and random state.

**Returns** A new instance of this component with identical parameters and random state.

**default_parameters**(*cls*)

Returns the default parameters for stacked ensemble classes.

**Returns** default parameters for this component.

**Return type** dict

**describe**(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

**Parameters**

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description as dictionary in the format {"name": name, "parameters": parameters}

**Returns** Returns dictionary if return_dict is True, else None.

**Return type** None or dict

**property feature_importance**(*self*)

Not implemented for StackedEnsembleClassifier and StackedEnsembleRegressor.

**fit**(*self*, *X: pandas.DataFrame*, *y: Optional[pandas.Series] = None*)

Fits estimator to data.

**Parameters**

- **X** (*pd.DataFrame*) – The input training data of shape [n_samples, n_features].
- **y** (*pd.Series, optional*) – The target training data of length [n_samples].

**Returns** self

**get_prediction_intervals**(*self*, *X: pandas.DataFrame*, *y: Optional[pandas.Series] = None*, *coverage: List[float] = None*, *predictions: pandas.Series = None*) → Dict[str, pandas.Series]

Find the prediction intervals using the fitted regressor.

This function takes the predictions of the fitted estimator and calculates the rolling standard deviation across all predictions using a window size of 5. The lower and upper predictions are determined by taking the percent point (quantile) function of the lower tail probability at each bound multiplied by the rolling standard deviation.

**Parameters**

- **X** (*pd.DataFrame*) – Data of shape [n_samples, n_features].
- **y** (*pd.Series*) – Target data. Ignored.
- **coverage** (*list[float]*) – A list of floats between the values 0 and 1 that the upper and lower bounds of the prediction interval should be calculated for.
- **predictions** (*pd.Series*) – Optional list of predictions to use. If None, will generate predictions using *X*.

**Returns** Prediction intervals, keys are in the format {coverage}_lower or {coverage}_upper.

**Return type** dict

**Raises** **MethodPropertyNotFoundError** – If the estimator does not support Time Series Regression as a problem type.

**static load**(*file_path*)

Loads component at file path.

**Parameters** **file_path** (*str*) – Location to load file.

**Returns** ComponentBase object

**needs_fitting**(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need to be fit or whose fit methods do nothing.

**Returns** True.

**property parameters**(*self*)

Returns the parameters which were used to initialize the component.

**predict**(*self*, *X: pandas.DataFrame*) → pandas.Series

Make predictions using selected features.

**Parameters** **X** (*pd.DataFrame*) – Data of shape [n_samples, n_features].

**Returns** Predicted values.

**Return type** pd.Series

**Raises** **MethodPropertyNotFoundError** – If estimator does not have a predict method or a component_obj that implements predict.

**predict_proba**(*self*, *X: pandas.DataFrame*) → pandas.Series

Make probability estimates for labels.

**Parameters** **X** (*pd.DataFrame*) – Features.

**Returns** Probability estimates.

**Return type** pd.Series

**Raises** **MethodPropertyNotFoundError** – If estimator does not have a predict_proba method or a component_obj that implements predict_proba.

**save**(*self*, *file_path*, *pickle_protocol=cloudpickle.DEFAULT_PROTOCOL*)

Saves component at file path.

**Parameters**

- **file_path** (*str*) – Location to save file.
- **pickle_protocol** (*int*) – The pickle data stream format.

**update_parameters**(*self*, *update_dict*, *reset_fit=True*)

Updates the parameter dictionary of the component.

**Parameters**

- **update_dict** (*dict*) – A dict of parameters to update.
- **reset_fit** (*bool, optional*) – If True, will set *_is_fitted* to False.

**class** evalml.pipelines.components.**StackedEnsembleRegressor**(*final_estimator=None*, *n_jobs=- 1*, *random_seed=0*, ***kwargs*)

Stacked Ensemble Regressor.

**Parameters**

- **final_estimator** (*Estimator or subclass*) – The regressor used to combine the base estimators. If None, uses ElasticNetRegressor.
- **n_jobs** (*int or None*) – Integer describing level of parallelism used for pipelines. None and 1 are equivalent. If set to -1, all CPUs are used. For n_jobs greater than -1, (n_cpus + 1 + n_jobs) are used. Defaults to -1. - Note: there could be some multi-process errors thrown for values of *n_jobs != 1*. If this is the case, please use *n_jobs = 1*.
- **random_seed** (*int*) – Seed for the random number generator. Defaults to 0.

### Example

```
>>> from evalml.pipelines.component_graph import ComponentGraph
>>> from evalml.pipelines.components.estimators.regressors.rf_regressor import␣
↪RandomForestRegressor
>>> from evalml.pipelines.components.estimators.regressors.elasticnet_regressor␣
↪import ElasticNetRegressor
...
>>> component_graph = {
...     "Random Forest": [RandomForestRegressor(random_seed=3), "X", "y"],
...     "Random Forest B": [RandomForestRegressor(random_seed=4), "X", "y"],
...     "Stacked Ensemble": [
...         StackedEnsembleRegressor(n_jobs=1, final_
↪estimator=RandomForestRegressor()),
```

(continues on next page)

(continued from previous page)

```
...         "Random Forest.x",
...         "Random Forest B.x",
...         "y",
...     ],
... }
...
>>> cg = ComponentGraph(component_graph)
>>> assert cg.default_parameters == {
...     'Random Forest Regressor': {'n_estimators': 100,
...                                 'max_depth': 6,
...                                 'n_jobs': -1},
...     'Stacked Ensemble Regressor': {'final_estimator': ElasticNetRegressor,
...                                    'n_jobs': -1}}
```

**Attributes**

| | |
|---|---|
| **hyper-parameter_ranges** | {} |
| **model_family** | ModelFamily.ENSEMBLE |
| **modifies_features** | True |
| **modifies_target** | False |
| **name** | Stacked Ensemble Regressor |
| **supported_problem_types** | [ ProblemTypes.REGRESSION, ProblemTypes.TIME_SERIES_REGRESSION,] |
| **training_only** | False |

**Methods**

| | |
|---|---|
| *clone* | Constructs a new component with the same parameters and random state. |
| *default_parameters* | Returns the default parameters for stacked ensemble classes. |
| *describe* | Describe a component and its parameters. |
| *feature_importance* | Not implemented for StackedEnsembleClassifier and StackedEnsembleRegressor. |
| *fit* | Fits estimator to data. |
| *get_prediction_intervals* | Find the prediction intervals using the fitted regressor. |
| *load* | Loads component at file path. |
| *needs_fitting* | Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances. |
| *parameters* | Returns the parameters which were used to initialize the component. |
| *predict* | Make predictions using selected features. |
| *predict_proba* | Make probability estimates for labels. |
| *save* | Saves component at file path. |
| *update_parameters* | Updates the parameter dictionary of the component. |

**clone**(*self*)

Constructs a new component with the same parameters and random state.

**Returns** A new instance of this component with identical parameters and random state.

**default_parameters**(*cls*)

Returns the default parameters for stacked ensemble classes.

**Returns** default parameters for this component.

**Return type** dict

**describe**(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

**Parameters**

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description as dictionary in the format {"name": name, "parameters": parameters}

**Returns** Returns dictionary if return_dict is True, else None.

**Return type** None or dict

**property feature_importance**(*self*)

Not implemented for StackedEnsembleClassifier and StackedEnsembleRegressor.

**fit**(*self*, *X: pandas.DataFrame*, *y: Optional[pandas.Series] = None*)

Fits estimator to data.

**Parameters**

- **X** (*pd.DataFrame*) – The input training data of shape [n_samples, n_features].
- **y** (*pd.Series, optional*) – The target training data of length [n_samples].

**Returns** self

**get_prediction_intervals**(*self*, *X: pandas.DataFrame*, *y: Optional[pandas.Series] = None*, *coverage: List[float] = None*, *predictions: pandas.Series = None*) → Dict[str, pandas.Series]

Find the prediction intervals using the fitted regressor.

This function takes the predictions of the fitted estimator and calculates the rolling standard deviation across all predictions using a window size of 5. The lower and upper predictions are determined by taking the percent point (quantile) function of the lower tail probability at each bound multiplied by the rolling standard deviation.

**Parameters**

- **X** (*pd.DataFrame*) – Data of shape [n_samples, n_features].
- **y** (*pd.Series*) – Target data. Ignored.
- **coverage** (*list[float]*) – A list of floats between the values 0 and 1 that the upper and lower bounds of the prediction interval should be calculated for.
- **predictions** (*pd.Series*) – Optional list of predictions to use. If None, will generate predictions using *X*.

**Returns** Prediction intervals, keys are in the format {coverage}_lower or {coverage}_upper.

**Return type** dict

**Raises** `MethodPropertyNotFoundError` – If the estimator does not support Time Series Regression as a problem type.

`static load`(*file_path*)

Loads component at file path.

**Parameters** `file_path` (*str*) – Location to load file.

**Returns** ComponentBase object

`needs_fitting`(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need to be fit or whose fit methods do nothing.

**Returns** True.

`property parameters`(*self*)

Returns the parameters which were used to initialize the component.

`predict`(*self*, *X: pandas.DataFrame*) → pandas.Series

Make predictions using selected features.

**Parameters** `X` (*pd.DataFrame*) – Data of shape [n_samples, n_features].

**Returns** Predicted values.

**Return type** pd.Series

**Raises** `MethodPropertyNotFoundError` – If estimator does not have a predict method or a component_obj that implements predict.

`predict_proba`(*self*, *X: pandas.DataFrame*) → pandas.Series

Make probability estimates for labels.

**Parameters** `X` (*pd.DataFrame*) – Features.

**Returns** Probability estimates.

**Return type** pd.Series

**Raises** `MethodPropertyNotFoundError` – If estimator does not have a predict_proba method or a component_obj that implements predict_proba.

`save`(*self*, *file_path*, *pickle_protocol=cloudpickle.DEFAULT_PROTOCOL*)

Saves component at file path.

**Parameters**

- `file_path` (*str*) – Location to save file.
- `pickle_protocol` (*int*) – The pickle data stream format.

`update_parameters`(*self*, *update_dict*, *reset_fit=True*)

Updates the parameter dictionary of the component.

**Parameters**

- `update_dict` (*dict*) – A dict of parameters to update.
- `reset_fit` (*bool, optional*) – If True, will set *_is_fitted* to False.

**class** evalml.pipelines.components.**StandardScaler**(*random_seed=0*, ***kwargs*)

A transformer that standardizes input features by removing the mean and scaling to unit variance.

**Parameters** **random_seed** (*int*) – Seed for the random number generator. Defaults to 0.

**Attributes**

| | |
|---|---|
| **hyper-parameter_ranges** | {} |
| **modifies_features** | True |
| **modifies_target** | False |
| **name** | Standard Scaler |
| **training_only** | False |

**Methods**

| | |
|---|---|
| *clone* | Constructs a new component with the same parameters and random state. |
| *default_parameters* | Returns the default parameters for this component. |
| *describe* | Describe a component and its parameters. |
| *fit* | Fits the standard scalar on the given data. |
| *fit_transform* | Fit and transform data using the standard scaler component. |
| *load* | Loads component at file path. |
| *needs_fitting* | Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances. |
| *parameters* | Returns the parameters which were used to initialize the component. |
| *save* | Saves component at file path. |
| *transform* | Transform data using the fitted standard scaler. |
| *update_parameters* | Updates the parameter dictionary of the component. |

**clone**(*self*)

Constructs a new component with the same parameters and random state.

**Returns** A new instance of this component with identical parameters and random state.

**default_parameters**(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

**Returns** Default parameters for this component.

**Return type** dict

**describe**(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

**Parameters**

- **print_name** (*bool, optional*) – whether to print name of component

- **return_dict** (*bool, optional*) – whether to return description as dictionary in the format {"name": name, "parameters": parameters}

**Returns** Returns dictionary if return_dict is True, else None.

**Return type** None or dict

**fit**(*self*, *X*, *y=None*)

Fits the standard scalar on the given data.

**Parameters**

- **X** (*pd.DataFrame*) – The input training data of shape [n_samples, n_features].
- **y** (*pd.Series, optional*) – The target training data of length [n_samples].

**Returns** self

**fit_transform**(*self*, *X*, *y=None*)

Fit and transform data using the standard scaler component.

**Parameters**

- **X** (*pd.DataFrame*) – The input training data of shape [n_samples, n_features].
- **y** (*pd.Series, optional*) – The target training data of length [n_samples].

**Returns** Transformed data.

**Return type** pd.DataFrame

**static load**(*file_path*)

Loads component at file path.

**Parameters** **file_path** (*str*) – Location to load file.

**Returns** ComponentBase object

**needs_fitting**(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need to be fit or whose fit methods do nothing.

**Returns** True.

**property parameters**(*self*)

Returns the parameters which were used to initialize the component.

**save**(*self*, *file_path*, *pickle_protocol=cloudpickle.DEFAULT_PROTOCOL*)

Saves component at file path.

**Parameters**

- **file_path** (*str*) – Location to save file.
- **pickle_protocol** (*int*) – The pickle data stream format.

**transform**(*self*, *X*, *y=None*)

Transform data using the fitted standard scaler.

**Parameters**

- **X** (*pd.DataFrame*) – The input training data of shape [n_samples, n_features].
- **y** (*pd.Series, optional*) – The target training data of length [n_samples].

**Returns** Transformed data.

**Return type** pd.DataFrame

**update_parameters**(*self*, *update_dict*, *reset_fit=True*)

Updates the parameter dictionary of the component.

**Parameters**

- **update_dict** (*dict*) – A dict of parameters to update.
- **reset_fit** (*bool, optional*) – If True, will set *_is_fitted* to False.

**class** evalml.pipelines.components.**STLDecomposer**(*time_index: str = None*, *degree: int = 1*, *period: int = None*, *seasonal_smoother: int = 7*, *random_seed: int = 0*, ***kwargs*)

Removes trends and seasonality from time series using the STL algorithm.

https://www.statsmodels.org/dev/generated/statsmodels.tsa.seasonal.STL.html

**Parameters**

- **time_index** (*str*) – Specifies the name of the column in X that provides the datetime objects. Defaults to None.
- **degree** (*int*) – Not currently used. STL 3x "degree-like" values. None are able to be set at this time. Defaults to 1.
- **period** (*int*) – The number of entries in the time series data that corresponds to one period of a cyclic signal. For instance, if data is known to possess a weekly seasonal signal, and if the data is daily data, the period should likely be 7. For daily data with a yearly seasonal signal, the period should likely be 365. If None, statsmodels will infer the period based on the frequency. Defaults to None.
- **seasonal_smoother** (*int*) – The length of the seasonal smoother used by the underlying STL algorithm. For compatibility, must be odd. If an even number is provided, the next, highest odd number will be used. Defaults to 7.
- **random_seed** (*int*) – Seed for the random number generator. Defaults to 0.

**Attributes**

| | |
|---|---|
| **hyper-parameter_ranges** | None |
| **invalid_frequencies** | [] |
| **modifies_features** | False |
| **modifies_target** | True |
| **name** | STL Decomposer |
| **needs_fitting** | True |
| **training_only** | False |

**Methods**

| | |
|---|---|
| clone | Constructs a new component with the same parameters and random state. |
| default_parameters | Returns the default parameters for this component. |
| describe | Describe a component and its parameters. |
| determine_periodicity | Function that uses autocorrelative methods to determine the likely most signficant period of the seasonal signal. |
| fit | Fits the STLDecomposer and determine the seasonal signal. |
| fit_transform | Removes fitted trend and seasonality from target variable. |
| get_trend_dataframe | Return a list of dataframes with 4 columns: signal, trend, seasonality, residual. |
| get_trend_prediction_intervals | Calculate the prediction intervals for the trend data. |
| inverse_transform | Adds back fitted trend and seasonality to target variable. |
| is_freq_valid | Determines if the given string represents a valid frequency for this decomposer. |
| load | Loads component at file path. |
| parameters | Returns the parameters which were used to initialize the component. |
| plot_decomposition | Plots the decomposition of the target signal. |
| save | Saves component at file path. |
| set_period | Function to set the component's seasonal period based on the target's seasonality. |
| transform | Transforms the target data by removing the STL trend and seasonality. |
| update_parameters | Updates the parameter dictionary of the component. |

**clone**(*self*)

Constructs a new component with the same parameters and random state.

**Returns** A new instance of this component with identical parameters and random state.

**default_parameters**(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

**Returns** Default parameters for this component.

**Return type** dict

**describe**(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

**Parameters**

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description as dictionary in the format {"name": name, "parameters": parameters}

**Returns** Returns dictionary if return_dict is True, else None.

**Return type** None or dict

**classmethod determine_periodicity**(*cls*, *X: pandas.DataFrame*, *y: pandas.Series*, *acf_threshold: float = 0.01*, *rel_max_order: int = 5*)

Function that uses autocorrelative methods to determine the likely most signficant period of the seasonal signal.

**Parameters**

- **X** (*pandas.DataFrame*) – The feature data of the time series problem.
- **y** (*pandas.Series*) – The target data of a time series problem.
- **acf_threshold** (*float*) – The threshold for the autocorrelation function to determine the period. Any values below the threshold are considered to be 0 and will not be considered for the period. Defaults to 0.01.
- **rel_max_order** (*int*) – The order of the relative maximum to determine the period. Defaults to 5.

**Returns**

**The integer number of entries in time series data over which the seasonal part of the target data** repeats. If the time series data is in days, then this is the number of days that it takes the target's seasonal signal to repeat. Note: the target data can contain multiple seasonal signals. This function will only return the stronger. E.g. if the target has both weekly and yearly seasonality, the function may return either "7" or "365", depending on which seasonality is more strongly autocorrelated. If no period is detected, returns None.

**Return type** int

**fit**(*self*, *X: pandas.DataFrame*, *y: pandas.Series = None*) → *STLDecomposer*

Fits the STLDecomposer and determine the seasonal signal.

Instantiates a statsmodels STL decompose object with the component's stored parameters and fits it. Since the statsmodels object does not fit the sklearn api, it is not saved during __init__() in _component_obj and will be re-instantiated each time fit is called.

To emulate the sklearn API, when the STL decomposer is fit, the full seasonal component, a single period sample of the seasonal component, the full trend-cycle component and the residual are saved.

y(t) = S(t) + T(t) + R(t)

**Parameters**

- **X** (*pd.DataFrame, optional*) – Conditionally used to build datetime index.
- **y** (*pd.Series*) – Target variable to detrend and deseasonalize.

**Returns** self

**Raises**

- **ValueError** – If y is None.
- **ValueError** – If target data doesn't have DatetimeIndex AND no Datetime features in features data

**fit_transform**(*self*, *X: pandas.DataFrame*, *y: pandas.Series = None*) → tuple[pandas.DataFrame, pandas.Series]

Removes fitted trend and seasonality from target variable.

**Parameters**

- **X** (*pd.DataFrame, optional*) – Ignored.
- **y** (*pd.Series*) – Target variable to detrend and deseasonalize.

**Returns**

**The first element are the input features returned without modification.** The second element is the target variable y with the fitted trend removed.

**Return type** tuple of pd.DataFrame, pd.Series

`get_trend_dataframe`(*self*, *X*, *y*)

Return a list of dataframes with 4 columns: signal, trend, seasonality, residual.

**Parameters**

- **X** (*pd.DataFrame*) – Input data with time series data in index.
- **y** (*pd.Series or pd.DataFrame*) – Target variable data provided as a Series for univariate problems or a DataFrame for multivariate problems.

**Returns**

**Each DataFrame contains the columns "signal", "trend", "seasonality" and "residual,"** with the latter 3 column values being the decomposed elements of the target data. The "signal" column is simply the input target signal but reindexed with a datetime index to match the input features.

**Return type** list of pd.DataFrame

**Raises**

- **TypeError** – If X does not have time-series data in the index.
- **ValueError** – If time series index of X does not have an inferred frequency.
- **ValueError** – If the forecaster associated with the detrender has not been fit yet.
- **TypeError** – If y is not provided as a pandas Series or DataFrame.

`get_trend_prediction_intervals`(*self*, *y*, *coverage=None*)

Calculate the prediction intervals for the trend data.

**Parameters**

- **y** (*pd.Series*) – Target data.
- **coverage** (*list[float]*) – A list of floats between the values 0 and 1 that the upper and lower bounds of the prediction interval should be calculated for.

**Returns** Prediction intervals, keys are in the format {coverage}_lower or {coverage}_upper.

**Return type** dict of pd.Series

`inverse_transform`(*self*, *y_t: pandas.Series*) → tuple[pandas.DataFrame, pandas.Series]

Adds back fitted trend and seasonality to target variable.

The STL trend is projected to cover the entire requested target range, then added back into the signal. Then, the seasonality is projected forward to and added back into the signal.

**Parameters** **y_t** (*pd.Series*) – Target variable.

**Returns**

**The first element are the input features returned without modification.** The second element is the target variable y with the trend and seasonality added back in.

**Return type** tuple of pd.DataFrame, pd.Series

**Raises** **ValueError** – If y is None.

**classmethod is_freq_valid**(*cls*, *freq: str*)

Determines if the given string represents a valid frequency for this decomposer.

**Parameters** **freq** (*str*) – A frequency to validate. See the pandas docs at https://pandas.pydata.org/pandas-docs/stable/user_guide/timeseries.html#offset-aliases for options.

**Returns** boolean representing whether the frequency is valid or not.

**static load**(*file_path*)

Loads component at file path.

**Parameters** **file_path** (*str*) – Location to load file.

**Returns** ComponentBase object

**property parameters**(*self*)

Returns the parameters which were used to initialize the component.

**plot_decomposition**(*self*, *X: pandas.DataFrame*, *y: pandas.Series*, *show: bool = False*) → tuple[matplotlib.pyplot.Figure, list]

Plots the decomposition of the target signal.

**Parameters**

- **X** (*pd.DataFrame*) – Input data with time series data in index.
- **y** (*pd.Series or pd.DataFrame*) – Target variable data provided as a Series for univariate problems or a DataFrame for multivariate problems.
- **show** (*bool*) – Whether to display the plot or not. Defaults to False.

**Returns**

**The figure and axes that have the decompositions** plotted on them

**Return type** matplotlib.pyplot.Figure, list[matplotlib.pyplot.Axes]

**save**(*self*, *file_path*, *pickle_protocol=cloudpickle.DEFAULT_PROTOCOL*)

Saves component at file path.

**Parameters**

- **file_path** (*str*) – Location to save file.
- **pickle_protocol** (*int*) – The pickle data stream format.

**set_period**(*self*, *X: pandas.DataFrame*, *y: pandas.Series*, *acf_threshold: float = 0.01*, *rel_max_order: int = 5*)

Function to set the component's seasonal period based on the target's seasonality.

**Parameters**

- **X** (*pandas.DataFrame*) – The feature data of the time series problem.
- **y** (*pandas.Series*) – The target data of a time series problem.
- **acf_threshold** (*float*) – The threshold for the autocorrelation function to determine the period. Any values below the threshold are considered to be 0 and will not be considered for the period. Defaults to 0.01.
- **rel_max_order** (*int*) – The order of the relative maximum to determine the period. Defaults to 5.

**transform**(*self*, *X: pandas.DataFrame*, *y: pandas.Series = None*) → tuple[pandas.DataFrame, pandas.Series]

Transforms the target data by removing the STL trend and seasonality.

Uses an ARIMA model to project forward the addititve trend and removes it. Then, utilizes the first period's worth of seasonal data determined in the .fit() function to extrapolate the seasonal signal of the data to be transformed. This seasonal signal is also assumed to be additive and is removed.

**Parameters**

- **X** (*pd.DataFrame, optional*) – Conditionally used to build datetime index.
- **y** (*pd.Series*) – Target variable to detrend and deseasonalize.

**Returns**

**The input features are returned without modification. The target** variable y is detrended and deseasonalized.

**Return type** tuple of pd.DataFrame, pd.Series

**Raises** **ValueError** – If target data doesn't have DatetimeIndex AND no Datetime features in features data

**update_parameters**(*self*, *update_dict*, *reset_fit=True*)

Updates the parameter dictionary of the component.

**Parameters**

- **update_dict** (*dict*) – A dict of parameters to update.
- **reset_fit** (*bool, optional*) – If True, will set *_is_fitted* to False.

**class** evalml.pipelines.components.**SVMClassifier**(*C=1.0*, *kernel='rbf'*, *gamma='auto'*, *probability=True*, *random_seed=0*, ***kwargs*)

Support Vector Machine Classifier.

**Parameters**

- **C** (*float*) – The regularization parameter. The strength of the regularization is inversely proportional to C. Must be strictly positive. The penalty is a squared l2 penalty. Defaults to 1.0.
- **kernel** (*{"poly", "rbf", "sigmoid"}*) – Specifies the kernel type to be used in the algorithm. Defaults to "rbf".
- **gamma** (*{"scale", "auto"} or float*) – Kernel coefficient for "rbf", "poly" and "sigmoid". Defaults to "auto". - If gamma='scale' is passed then it uses 1 / (n_features * X.var()) as value of gamma - If "auto" (default), uses 1 / n_features
- **probability** (*boolean*) – Whether to enable probability estimates. Defaults to True.
- **random_seed** (*int*) – Seed for the random number generator. Defaults to 0.

**Attributes**

| | |
|---|---|
| **hyper-parameter_ranges** | { "C": Real(0, 10), "kernel": ["poly", "rbf", "sigmoid"], "gamma": ["scale", "auto"],} |
| **model_family** | ModelFamily.SVM |
| **modifies_features** | True |
| **modifies_target** | False |
| **name** | SVM Classifier |
| **supported_problem_types** | [ ProblemTypes.BINARY, ProblemTypes.MULTICLASS, ProblemTypes.TIME_SERIES_BINARY, ProblemTypes.TIME_SERIES_MULTICLASS,] |
| **training_only** | False |

**Methods**

| | |
|---|---|
| *clone* | Constructs a new component with the same parameters and random state. |
| *default_parameters* | Returns the default parameters for this component. |
| *describe* | Describe a component and its parameters. |
| *feature_importance* | Feature importance only works with linear kernels. |
| *fit* | Fits estimator to data. |
| *get_prediction_intervals* | Find the prediction intervals using the fitted regressor. |
| *load* | Loads component at file path. |
| *needs_fitting* | Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances. |
| *parameters* | Returns the parameters which were used to initialize the component. |
| *predict* | Make predictions using selected features. |
| *predict_proba* | Make probability estimates for labels. |
| *save* | Saves component at file path. |
| *update_parameters* | Updates the parameter dictionary of the component. |

**clone**(*self*)

Constructs a new component with the same parameters and random state.

**Returns** A new instance of this component with identical parameters and random state.

**default_parameters**(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

**Returns** Default parameters for this component.

**Return type** dict

**describe**(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

**Parameters**

- **print_name** (*bool, optional*) – whether to print name of component

- **return_dict** (*bool, optional*) – whether to return description as dictionary in the format {"name": name, "parameters": parameters}

**Returns** Returns dictionary if return_dict is True, else None.

**Return type** None or dict

**property feature_importance**(*self*)

Feature importance only works with linear kernels.

If the kernel isn't linear, we return a numpy array of zeros.

**Returns** Feature importance of fitted SVM classifier or a numpy array of zeroes if the kernel is not linear.

**fit**(*self*, *X: pandas.DataFrame*, *y: Optional[pandas.Series] = None*)

Fits estimator to data.

**Parameters**

- **X** (*pd.DataFrame*) – The input training data of shape [n_samples, n_features].
- **y** (*pd.Series, optional*) – The target training data of length [n_samples].

**Returns** self

**get_prediction_intervals**(*self*, *X: pandas.DataFrame*, *y: Optional[pandas.Series] = None*, *coverage: List[float] = None*, *predictions: pandas.Series = None*) → Dict[str, pandas.Series]

Find the prediction intervals using the fitted regressor.

This function takes the predictions of the fitted estimator and calculates the rolling standard deviation across all predictions using a window size of 5. The lower and upper predictions are determined by taking the percent point (quantile) function of the lower tail probability at each bound multiplied by the rolling standard deviation.

**Parameters**

- **X** (*pd.DataFrame*) – Data of shape [n_samples, n_features].
- **y** (*pd.Series*) – Target data. Ignored.
- **coverage** (*list[float]*) – A list of floats between the values 0 and 1 that the upper and lower bounds of the prediction interval should be calculated for.
- **predictions** (*pd.Series*) – Optional list of predictions to use. If None, will generate predictions using *X*.

**Returns** Prediction intervals, keys are in the format {coverage}_lower or {coverage}_upper.

**Return type** dict

**Raises** **MethodPropertyNotFoundError** – If the estimator does not support Time Series Regression as a problem type.

**static load**(*file_path*)

Loads component at file path.

**Parameters** **file_path** (*str*) – Location to load file.

**Returns** ComponentBase object

**needs_fitting**(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need to be fit or whose fit methods do nothing.

**Returns** True.

**property parameters**(*self*)

Returns the parameters which were used to initialize the component.

**predict**(*self*, *X: pandas.DataFrame*) → pandas.Series

Make predictions using selected features.

**Parameters** **X** (*pd.DataFrame*) – Data of shape [n_samples, n_features].

**Returns** Predicted values.

**Return type** pd.Series

**Raises** **MethodPropertyNotFoundError** – If estimator does not have a predict method or a component_obj that implements predict.

**predict_proba**(*self*, *X: pandas.DataFrame*) → pandas.Series

Make probability estimates for labels.

**Parameters** **X** (*pd.DataFrame*) – Features.

**Returns** Probability estimates.

**Return type** pd.Series

**Raises** **MethodPropertyNotFoundError** – If estimator does not have a predict_proba method or a component_obj that implements predict_proba.

**save**(*self*, *file_path*, *pickle_protocol=cloudpickle.DEFAULT_PROTOCOL*)

Saves component at file path.

**Parameters**

- **file_path** (*str*) – Location to save file.
- **pickle_protocol** (*int*) – The pickle data stream format.

**update_parameters**(*self*, *update_dict*, *reset_fit=True*)

Updates the parameter dictionary of the component.

**Parameters**

- **update_dict** (*dict*) – A dict of parameters to update.
- **reset_fit** (*bool, optional*) – If True, will set *_is_fitted* to False.

**class** evalml.pipelines.components.**SVMRegressor**(*C=1.0*, *kernel='rbf'*, *gamma='auto'*, *random_seed=0*, ***kwargs*)

Support Vector Machine Regressor.

**Parameters**

- **C** (*float*) – The regularization parameter. The strength of the regularization is inversely proportional to C. Must be strictly positive. The penalty is a squared l2 penalty. Defaults to 1.0.
- **kernel** (*{"poly", "rbf", "sigmoid"}*) – Specifies the kernel type to be used in the algorithm. Defaults to "rbf".

- **gamma** (*{"scale", "auto"} or float*) – Kernel coefficient for "rbf", "poly" and "sigmoid". Defaults to "auto". - If gamma='scale' is passed then it uses 1 / (n_features * X.var()) as value of gamma - If "auto" (default), uses 1 / n_features
- **random_seed** (*int*) – Seed for the random number generator. Defaults to 0.

**Attributes**

| | |
|---|---|
| **hyper-parameter_ranges** | { "C": Real(0, 10), "kernel": ["poly", "rbf", "sigmoid"], "gamma": ["scale", "auto"],} |
| **model_family** | ModelFamily.SVM |
| **modifies_features** | True |
| **modifies_target** | False |
| **name** | SVM Regressor |
| **supported_problem_types** | [ ProblemTypes.REGRESSION, ProblemTypes.TIME_SERIES_REGRESSION,] |
| **training_only** | False |

**Methods**

| | |
|---|---|
| *clone* | Constructs a new component with the same parameters and random state. |
| *default_parameters* | Returns the default parameters for this component. |
| *describe* | Describe a component and its parameters. |
| *feature_importance* | Feature importance of fitted SVM regresor. |
| *fit* | Fits estimator to data. |
| *get_prediction_intervals* | Find the prediction intervals using the fitted regressor. |
| *load* | Loads component at file path. |
| *needs_fitting* | Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances. |
| *parameters* | Returns the parameters which were used to initialize the component. |
| *predict* | Make predictions using selected features. |
| *predict_proba* | Make probability estimates for labels. |
| *save* | Saves component at file path. |
| *update_parameters* | Updates the parameter dictionary of the component. |

**clone**(*self*)

Constructs a new component with the same parameters and random state.

**Returns** A new instance of this component with identical parameters and random state.

**default_parameters**(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

**Returns** Default parameters for this component.

**Return type** dict

**describe**(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

**Parameters**

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description as dictionary in the format {"name": name, "parameters": parameters}

**Returns** Returns dictionary if return_dict is True, else None.

**Return type** None or dict

**property feature_importance**(*self*)

Feature importance of fitted SVM regresor.

Only works with linear kernels. If the kernel isn't linear, we return a numpy array of zeros.

**Returns** The feature importance of the fitted SVM regressor, or an array of zeroes if the kernel is not linear.

**fit**(*self*, *X: pandas.DataFrame*, *y: Optional[pandas.Series] = None*)

Fits estimator to data.

**Parameters**

- **X** (*pd.DataFrame*) – The input training data of shape [n_samples, n_features].
- **y** (*pd.Series, optional*) – The target training data of length [n_samples].

**Returns** self

**get_prediction_intervals**(*self*, *X: pandas.DataFrame*, *y: Optional[pandas.Series] = None*, *coverage: List[float] = None*, *predictions: pandas.Series = None*) → Dict[str, pandas.Series]

Find the prediction intervals using the fitted regressor.

This function takes the predictions of the fitted estimator and calculates the rolling standard deviation across all predictions using a window size of 5. The lower and upper predictions are determined by taking the percent point (quantile) function of the lower tail probability at each bound multiplied by the rolling standard deviation.

**Parameters**

- **X** (*pd.DataFrame*) – Data of shape [n_samples, n_features].
- **y** (*pd.Series*) – Target data. Ignored.
- **coverage** (*list[float]*) – A list of floats between the values 0 and 1 that the upper and lower bounds of the prediction interval should be calculated for.
- **predictions** (*pd.Series*) – Optional list of predictions to use. If None, will generate predictions using *X*.

**Returns** Prediction intervals, keys are in the format {coverage}_lower or {coverage}_upper.

**Return type** dict

**Raises** **MethodPropertyNotFoundError** – If the estimator does not support Time Series Regression as a problem type.

**static load**(*file_path*)

Loads component at file path.

**Parameters** **file_path** (*str*) – Location to load file.

**Returns** ComponentBase object

**needs_fitting**(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need to be fit or whose fit methods do nothing.

**Returns** True.

**property parameters**(*self*)

Returns the parameters which were used to initialize the component.

**predict**(*self*, *X: pandas.DataFrame*) → pandas.Series

Make predictions using selected features.

**Parameters** **X** (*pd.DataFrame*) – Data of shape [n_samples, n_features].

**Returns** Predicted values.

**Return type** pd.Series

**Raises** **MethodPropertyNotFoundError** – If estimator does not have a predict method or a component_obj that implements predict.

**predict_proba**(*self*, *X: pandas.DataFrame*) → pandas.Series

Make probability estimates for labels.

**Parameters** **X** (*pd.DataFrame*) – Features.

**Returns** Probability estimates.

**Return type** pd.Series

**Raises** **MethodPropertyNotFoundError** – If estimator does not have a predict_proba method or a component_obj that implements predict_proba.

**save**(*self*, *file_path*, *pickle_protocol=cloudpickle.DEFAULT_PROTOCOL*)

Saves component at file path.

**Parameters**

- **file_path** (*str*) – Location to save file.
- **pickle_protocol** (*int*) – The pickle data stream format.

**update_parameters**(*self*, *update_dict*, *reset_fit=True*)

Updates the parameter dictionary of the component.

**Parameters**

- **update_dict** (*dict*) – A dict of parameters to update.
- **reset_fit** (*bool, optional*) – If True, will set *_is_fitted* to False.

**class** evalml.pipelines.components.**TargetEncoder**(*cols=None*, *smoothing=1*, *handle_unknown='value'*, *handle_missing='value'*, *random_seed=0*, ***kwargs*)

A transformer that encodes categorical features into target encodings.

**Parameters**

- **cols** (*list*) – Columns to encode. If None, all string columns will be encoded, otherwise only the columns provided will be encoded. Defaults to None
- **smoothing** (*float*) – The smoothing factor to apply. The larger this value is, the more influence the expected target value has on the resulting target encodings. Must be strictly larger than 0. Defaults to 1.0
- **handle_unknown** (*string*) – Determines how to handle unknown categories for a feature encountered. Options are 'value', 'error', nd 'return_nan'. Defaults to 'value', which replaces with the target mean
- **handle_missing** (*string*) – Determines how to handle missing values encountered during *fit* or *transform*. Options are 'value', 'error', and 'return_nan'. Defaults to 'value', which replaces with the target mean
- **random_seed** (*int*) – Seed for the random number generator. Defaults to 0.

**Attributes**

| | |
|---|---|
| **hyper-parameter_ranges** | {} |
| **modifies_features** | True |
| **modifies_target** | False |
| **name** | Target Encoder |
| **training_only** | False |

**Methods**

| | |
|---|---|
| *clone* | Constructs a new component with the same parameters and random state. |
| *default_parameters* | Returns the default parameters for this component. |
| *describe* | Describe a component and its parameters. |
| *fit* | Fits the target encoder. |
| *fit_transform* | Fit and transform data using the target encoder. |
| *get_feature_names* | Return feature names for the input features after fitting. |
| *load* | Loads component at file path. |
| *needs_fitting* | Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances. |
| *parameters* | Returns the parameters which were used to initialize the component. |
| *save* | Saves component at file path. |
| *transform* | Transform data using the fitted target encoder. |
| *update_parameters* | Updates the parameter dictionary of the component. |

**clone**(*self*)

Constructs a new component with the same parameters and random state.

**Returns** A new instance of this component with identical parameters and random state.

**default_parameters**(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

**Returns** Default parameters for this component.

**Return type** dict

**describe**(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

**Parameters**

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description as dictionary in the format {"name": name, "parameters": parameters}

**Returns** Returns dictionary if return_dict is True, else None.

**Return type** None or dict

**fit**(*self*, *X*, *y*)

Fits the target encoder.

**Parameters**

- **X** (*pd.DataFrame*) – The input training data of shape [n_samples, n_features].
- **y** (*pd.Series, optional*) – The target training data of length [n_samples].

**Returns** self

**fit_transform**(*self*, *X*, *y*)

Fit and transform data using the target encoder.

**Parameters**

- **X** (*pd.DataFrame*) – The input training data of shape [n_samples, n_features].
- **y** (*pd.Series, optional*) – The target training data of length [n_samples].

**Returns** Transformed data.

**Return type** pd.DataFrame

**get_feature_names**(*self*)

Return feature names for the input features after fitting.

**Returns** The feature names after encoding.

**Return type** np.array

**static load**(*file_path*)

Loads component at file path.

**Parameters** **file_path** (*str*) – Location to load file.

**Returns** ComponentBase object

**needs_fitting**(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need to be fit or whose fit methods do nothing.

**Returns** True.

**property parameters**(*self*)

Returns the parameters which were used to initialize the component.

**save**(*self*, *file_path*, *pickle_protocol=cloudpickle.DEFAULT_PROTOCOL*)

Saves component at file path.

**Parameters**

- **file_path** (*str*) – Location to save file.
- **pickle_protocol** (*int*) – The pickle data stream format.

**transform**(*self*, *X*, *y=None*)

Transform data using the fitted target encoder.

**Parameters**

- **X** (*pd.DataFrame*) – The input training data of shape [n_samples, n_features].
- **y** (*pd.Series, optional*) – The target training data of length [n_samples].

**Returns** Transformed data.

**Return type** pd.DataFrame

**update_parameters**(*self*, *update_dict*, *reset_fit=True*)

Updates the parameter dictionary of the component.

**Parameters**

- **update_dict** (*dict*) – A dict of parameters to update.
- **reset_fit** (*bool, optional*) – If True, will set *_is_fitted* to False.

**class** evalml.pipelines.components.**TargetImputer**(*impute_strategy='most_frequent'*, *fill_value=None*, *random_seed=0*, ***kwargs*)

Imputes missing target data according to a specified imputation strategy.

**Parameters**

- **impute_strategy** (*string*) – Impute strategy to use. Valid values include "mean", "median", "most_frequent", "constant" for numerical data, and "most_frequent", "constant" for object data types. Defaults to "most_frequent".
- **fill_value** (*string*) – When impute_strategy == "constant", fill_value is used to replace missing data. Defaults to None which uses 0 when imputing numerical data and "missing_value" for strings or object data types.
- **random_seed** (*int*) – Seed for the random number generator. Defaults to 0.

**Attributes**

| | |
|---|---|
| **hyper-parameter_ranges** | { "impute_strategy": ["mean", "median", "most_frequent"]} |
| **modifies_features** | False |
| **modifies_target** | True |
| **name** | Target Imputer |
| **training_only** | False |

**Methods**

| | |
|---|---|
| *clone* | Constructs a new component with the same parameters and random state. |
| *default_parameters* | Returns the default parameters for this component. |
| *describe* | Describe a component and its parameters. |
| *fit* | Fits imputer to target data. 'None' values are converted to np.nan before imputation and are treated as the same. |
| *fit_transform* | Fits on and transforms the input target data. |
| *load* | Loads component at file path. |
| *needs_fitting* | Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances. |
| *parameters* | Returns the parameters which were used to initialize the component. |
| *save* | Saves component at file path. |
| *transform* | Transforms input target data by imputing missing values. 'None' and np.nan values are treated as the same. |
| *update_parameters* | Updates the parameter dictionary of the component. |

**clone**(*self*)

Constructs a new component with the same parameters and random state.

**Returns** A new instance of this component with identical parameters and random state.

**default_parameters**(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

**Returns** Default parameters for this component.

**Return type** dict

**describe**(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

**Parameters**

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description as dictionary in the format {"name": name, "parameters": parameters}

**Returns** Returns dictionary if return_dict is True, else None.

**Return type** None or dict

`fit`(*self*, *X*, *y*)

Fits imputer to target data. 'None' values are converted to np.nan before imputation and are treated as the same.

**Parameters**

- **X** (*pd.DataFrame or np.ndarray*) – The input training data of shape [n_samples, n_features]. Ignored.
- **y** (*pd.Series, optional*) – The target training data of length [n_samples].

**Returns** self

**Raises** **TypeError** – If target is filled with all null values.

`fit_transform`(*self*, *X*, *y*)

Fits on and transforms the input target data.

**Parameters**

- **X** (*pd.DataFrame*) – Features. Ignored.
- **y** (*pd.Series*) – Target data to impute.

**Returns** The original X, transformed y

**Return type** (pd.DataFrame, pd.Series)

`static load`(*file_path*)

Loads component at file path.

**Parameters** **file_path** (*str*) – Location to load file.

**Returns** ComponentBase object

`needs_fitting`(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need to be fit or whose fit methods do nothing.

**Returns** True.

`property parameters`(*self*)

Returns the parameters which were used to initialize the component.

`save`(*self*, *file_path*, *pickle_protocol=cloudpickle.DEFAULT_PROTOCOL*)

Saves component at file path.

**Parameters**

- **file_path** (*str*) – Location to save file.
- **pickle_protocol** (*int*) – The pickle data stream format.

`transform`(*self*, *X*, *y*)

Transforms input target data by imputing missing values. 'None' and np.nan values are treated as the same.

**Parameters**

- **X** (*pd.DataFrame*) – Features. Ignored.

- **y** (*pd.Series*) – Target data to impute.

**Returns** The original X, transformed y

**Return type** (pd.DataFrame, pd.Series)

**update_parameters**(*self*, *update_dict*, *reset_fit=True*)

Updates the parameter dictionary of the component.

**Parameters**

- **update_dict** (*dict*) – A dict of parameters to update.
- **reset_fit** (*bool, optional*) – If True, will set *_is_fitted* to False.

**class** evalml.pipelines.components.**TimeSeriesBaselineEstimator**(*gap=1*, *forecast_horizon=1*, *random_seed=0*, ***kwargs*)

Time series estimator that predicts using the naive forecasting approach.

This is useful as a simple baseline estimator for time series problems.

**Parameters**

- **gap** (*int*) – Gap between prediction date and target date and must be a positive integer. If gap is 0, target date will be shifted ahead by 1 time period. Defaults to 1.
- **forecast_horizon** (*int*) – Number of time steps the model is expected to predict.
- **random_seed** (*int*) – Seed for the random number generator. Defaults to 0.

**Attributes**

| | |
|---|---|
| **hyper-parameter_ranges** | {} |
| **model_family** | ModelFamily.BASELINE |
| **modifies_features** | True |
| **modifies_target** | False |
| **name** | Time Series Baseline Estimator |
| **supported_problem_types** | [ ProblemTypes.TIME_SERIES_REGRESSION, ProblemTypes.TIME_SERIES_BINARY, ProblemTypes.TIME_SERIES_MULTICLASS,] |
| **training_only** | False |

**Methods**

| | |
|---|---|
| *clone* | Constructs a new component with the same parameters and random state. |
| *default_parameters* | Returns the default parameters for this component. |
| *describe* | Describe a component and its parameters. |
| *feature_importance* | Returns importance associated with each feature. |
| *fit* | Fits time series baseline estimator to data. |
| *get_prediction_intervals* | Find the prediction intervals using the fitted regressor. |
| *load* | Loads component at file path. |
| *needs_fitting* | Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances. |
| *parameters* | Returns the parameters which were used to initialize the component. |
| *predict* | Make predictions using fitted time series baseline estimator. |
| *predict_proba* | Make prediction probabilities using fitted time series baseline estimator. |
| *save* | Saves component at file path. |
| *update_parameters* | Updates the parameter dictionary of the component. |

**clone**(*self*)

Constructs a new component with the same parameters and random state.

**Returns** A new instance of this component with identical parameters and random state.

**default_parameters**(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

**Returns** Default parameters for this component.

**Return type** dict

**describe**(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

**Parameters**

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description as dictionary in the format {"name": name, "parameters": parameters}

**Returns** Returns dictionary if return_dict is True, else None.

**Return type** None or dict

**property feature_importance**(*self*)

Returns importance associated with each feature.

Since baseline estimators do not use input features to calculate predictions, returns an array of zeroes.

**Returns** An array of zeroes.

**Return type** np.ndarray (float)

**fit**(*self*, *X*, *y=None*)

Fits time series baseline estimator to data.

**Parameters**

- **X** (*pd.DataFrame*) – The input training data of shape [n_samples, n_features].
- **y** (*pd.Series*) – The target training data of length [n_samples].

**Returns** self

**Raises** **ValueError** – If input y is None.

**get_prediction_intervals**(*self*, *X: pandas.DataFrame*, *y: Optional[pandas.Series] = None*, *coverage: List[float] = None*, *predictions: pandas.Series = None*) → Dict[str, pandas.Series]

Find the prediction intervals using the fitted regressor.

This function takes the predictions of the fitted estimator and calculates the rolling standard deviation across all predictions using a window size of 5. The lower and upper predictions are determined by taking the percent point (quantile) function of the lower tail probability at each bound multiplied by the rolling standard deviation.

**Parameters**

- **X** (*pd.DataFrame*) – Data of shape [n_samples, n_features].
- **y** (*pd.Series*) – Target data. Ignored.
- **coverage** (*list[float]*) – A list of floats between the values 0 and 1 that the upper and lower bounds of the prediction interval should be calculated for.
- **predictions** (*pd.Series*) – Optional list of predictions to use. If None, will generate predictions using *X*.

**Returns** Prediction intervals, keys are in the format {coverage}_lower or {coverage}_upper.

**Return type** dict

**Raises** **MethodPropertyNotFoundError** – If the estimator does not support Time Series Regression as a problem type.

**static load**(*file_path*)

Loads component at file path.

**Parameters** **file_path** (*str*) – Location to load file.

**Returns** ComponentBase object

**needs_fitting**(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need to be fit or whose fit methods do nothing.

**Returns** True.

**property parameters**(*self*)

Returns the parameters which were used to initialize the component.

**predict**(*self*, *X*)

Make predictions using fitted time series baseline estimator.

**Parameters** **X** (*pd.DataFrame*) – Data of shape [n_samples, n_features].

**Returns** Predicted values.

**Return type** pd.Series

**Raises** `ValueError` – If input y is None.

`predict_proba`(*self*, *X*)

Make prediction probabilities using fitted time series baseline estimator.

**Parameters** `X` (*pd.DataFrame*) – Data of shape [n_samples, n_features].

**Returns** Predicted probability values.

**Return type** pd.DataFrame

**Raises** `ValueError` – If input y is None.

`save`(*self*, *file_path*, *pickle_protocol=cloudpickle.DEFAULT_PROTOCOL*)

Saves component at file path.

**Parameters**

- `file_path` (*str*) – Location to save file.
- `pickle_protocol` (*int*) – The pickle data stream format.

`update_parameters`(*self*, *update_dict*, *reset_fit=True*)

Updates the parameter dictionary of the component.

**Parameters**

- `update_dict` (*dict*) – A dict of parameters to update.
- `reset_fit` (*bool, optional*) – If True, will set *_is_fitted* to False.

`class evalml.pipelines.components.TimeSeriesFeaturizer`(*time_index=None*, *max_delay=2*, *gap=0*, *forecast_horizon=1*, *conf_level=0.05*, *rolling_window_size=0.25*, *delay_features=True*, *delay_target=True*, *random_seed=0*, ***kwargs*)

Transformer that delays input features and target variable for time series problems.

This component uses an algorithm based on the autocorrelation values of the target variable to determine which lags to select from the set of all possible lags.

The algorithm is based on the idea that the local maxima of the autocorrelation function indicate the lags that have the most impact on the present time.

The algorithm computes the autocorrelation values and finds the local maxima, called "peaks", that are significant at the given conf_level. Since lags in the range [0, 10] tend to be predictive but not local maxima, the union of the peaks is taken with the significant lags in the range [0, 10]. At the end, only selected lags in the range [0, max_delay] are used.

Parametrizing the algorithm by conf_level lets the AutoMLAlgorithm tune the set of lags chosen so that the chances of finding a good set of lags is higher.

Using conf_level value of 1 selects all possible lags.

**Parameters**

- `time_index` (*str*) – Name of the column containing the datetime information used to order the data. Ignored.
- `max_delay` (*int*) – Maximum number of time units to delay each feature. Defaults to 2.

- **forecast_horizon** (*int*) – The number of time periods the pipeline is expected to forecast.
- **conf_level** (*float*) – Float in range (0, 1] that determines the confidence interval size used to select which lags to compute from the set of [1, max_delay]. A delay of 1 will always be computed. If 1, selects all possible lags in the set of [1, max_delay], inclusive.
- **rolling_window_size** (*float*) – Float in range (0, 1] that determines the size of the window used for rolling features. Size is computed as rolling_window_size * max_delay.
- **delay_features** (*bool*) – Whether to delay the input features. Defaults to True.
- **delay_target** (*bool*) – Whether to delay the target. Defaults to True.
- **gap** (*int*) – The number of time units between when the features are collected and when the target is collected. For example, if you are predicting the next time step's target, gap=1. This is only needed because when gap=0, we need to be sure to start the lagging of the target variable at 1. Defaults to 1.
- **random_seed** (*int*) – Seed for the random number generator. This transformer performs the same regardless of the random seed provided.

**Attributes**

| | |
|---|---|
| **hyper-parameter_ranges** | Real(0.001, 1.0), "rolling_window_size": Real(0.001, 1.0)}:type: {"conf_level" |
| **modifies_features** | True |
| **modifies_target** | False |
| **name** | Time Series Featurizer |
| **needs_fitting** | True |
| **target_colname_prefix** | target_delay_{} |
| **training_only** | False |

**Methods**

| | |
|---|---|
| *clone* | Constructs a new component with the same parameters and random state. |
| *default_parameters* | Returns the default parameters for this component. |
| *describe* | Describe a component and its parameters. |
| *fit* | Fits the DelayFeatureTransformer. |
| *fit_transform* | Fit the component and transform the input data. |
| *load* | Loads component at file path. |
| *parameters* | Returns the parameters which were used to initialize the component. |
| *save* | Saves component at file path. |
| *transform* | Computes the delayed values and rolling means for X and y. |
| *update_parameters* | Updates the parameter dictionary of the component. |

**clone**(*self*)

Constructs a new component with the same parameters and random state.

**Returns** A new instance of this component with identical parameters and random state.

`default_parameters`(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

**Returns** Default parameters for this component.

**Return type** dict

`describe`(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

**Parameters**

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description as dictionary in the format {"name": name, "parameters": parameters}

**Returns** Returns dictionary if return_dict is True, else None.

**Return type** None or dict

`fit`(*self*, *X*, *y=None*)

Fits the DelayFeatureTransformer.

**Parameters**

- **X** (*pd.DataFrame or np.ndarray*) – The input training data of shape [n_samples, n_features]
- **y** (*pd.Series, optional*) – The target training data of length [n_samples]

**Returns** self

**Raises** **ValueError** – if self.time_index is None

`fit_transform`(*self*, *X*, *y=None*)

Fit the component and transform the input data.

**Parameters**

- **X** (*pd.DataFrame*) – Data to transform.
- **y** (*pd.Series, or None*) – Target.

**Returns** Transformed X.

**Return type** pd.DataFrame

`static load`(*file_path*)

Loads component at file path.

**Parameters** **file_path** (*str*) – Location to load file.

**Returns** ComponentBase object

`property parameters`(*self*)

Returns the parameters which were used to initialize the component.

`save`(*self*, *file_path*, *pickle_protocol=cloudpickle.DEFAULT_PROTOCOL*)

Saves component at file path.

**Parameters**

- **file_path** (*str*) – Location to save file.
- **pickle_protocol** (*int*) – The pickle data stream format.

**transform**(*self*, *X*, *y=None*)

Computes the delayed values and rolling means for X and y.

The chosen delays are determined by the autocorrelation function of the target variable. See the class docstring for more information on how they are chosen. If y is None, all possible lags are chosen.

If y is not None, it will also compute the delayed values for the target variable.

The rolling means for all numeric features in X and y, if y is numeric, are also returned.

**Parameters**

- **X** (*pd.DataFrame or None*) – Data to transform. None is expected when only the target variable is being used.
- **y** (*pd.Series, or None*) – Target.

**Returns** Transformed X. No original features are returned.

**Return type** pd.DataFrame

**update_parameters**(*self*, *update_dict*, *reset_fit=True*)

Updates the parameter dictionary of the component.

**Parameters**

- **update_dict** (*dict*) – A dict of parameters to update.
- **reset_fit** (*bool, optional*) – If True, will set *_is_fitted* to False.

**class** evalml.pipelines.components.**TimeSeriesImputer**(*categorical_impute_strategy='forwards_fill'*, *numeric_impute_strategy='interpolate'*, *target_impute_strategy='forwards_fill'*, *random_seed=0*, ***kwargs*)

Imputes missing data according to a specified timeseries-specific imputation strategy.

This Transformer should be used after the *TimeSeriesRegularizer* in order to impute the missing values that were added to X and y (if passed).

**Parameters**

- **categorical_impute_strategy** (*string*) – Impute strategy to use for string, object, boolean, categorical dtypes. Valid values include "backwards_fill" and "forwards_fill". Defaults to "forwards_fill".
- **numeric_impute_strategy** (*string*) – Impute strategy to use for numeric columns. Valid values include "backwards_fill", "forwards_fill", and "interpolate". Defaults to "interpolate".
- **target_impute_strategy** (*string*) – Impute strategy to use for the target column. Valid values include "backwards_fill", "forwards_fill", and "interpolate". Defaults to "forwards_fill".
- **random_seed** (*int*) – Seed for the random number generator. Defaults to 0.

**Raises** **ValueError** – If categorical_impute_strategy, numeric_impute_strategy, or target_impute_strategy is not one of the valid values.

**Attributes**

| | |
|---|---|
| **hyper-parameter_ranges** | { "categorical_impute_strategy": ["backwards_fill", "forwards_fill"], "numeric_impute_strategy": ["backwards_fill", "forwards_fill", "interpolate"], "target_impute_strategy": ["backwards_fill", "forwards_fill", "interpolate"],} |
| **modifies_features** | True |
| **modifies_target** | True |
| **name** | Time Series Imputer |
| **training_only** | True |

**Methods**

| | |
|---|---|
| *clone* | Constructs a new component with the same parameters and random state. |
| *default_parameters* | Returns the default parameters for this component. |
| *describe* | Describe a component and its parameters. |
| *fit* | Fits imputer to data. |
| *fit_transform* | Fits on X and transforms X. |
| *load* | Loads component at file path. |
| *needs_fitting* | Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances. |
| *parameters* | Returns the parameters which were used to initialize the component. |
| *save* | Saves component at file path. |
| *transform* | Transforms data X by imputing missing values using specified timeseries-specific strategies. 'None' values are converted to np.nan before imputation and are treated as the same. |
| *update_parameters* | Updates the parameter dictionary of the component. |

**clone**(*self*)

Constructs a new component with the same parameters and random state.

**Returns** A new instance of this component with identical parameters and random state.

**default_parameters**(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

**Returns** Default parameters for this component.

**Return type** dict

**describe**(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

**Parameters**

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description as dictionary in the format {"name": name, "parameters": parameters}

**Returns** Returns dictionary if return_dict is True, else None.

**Return type** None or dict

`fit`(*self*, *X*, *y=None*)

Fits imputer to data.

'None' values are converted to np.nan before imputation and are treated as the same. If a value is missing at the beginning or end of a column, that value will be imputed using backwards fill or forwards fill as necessary, respectively.

**Parameters**

- **X** (*pd.DataFrame, np.ndarray*) – The input training data of shape [n_samples, n_features]
- **y** (*pd.Series, optional*) – The target training data of length [n_samples]

**Returns** self

`fit_transform`(*self*, *X*, *y=None*)

Fits on X and transforms X.

**Parameters**

- **X** (*pd.DataFrame*) – Data to fit and transform.
- **y** (*pd.Series*) – Target data.

**Returns** Transformed X.

**Return type** pd.DataFrame

**Raises** `MethodPropertyNotFoundError` – If transformer does not have a transform method or a component_obj that implements transform.

`static load`(*file_path*)

Loads component at file path.

**Parameters** **file_path** (*str*) – Location to load file.

**Returns** ComponentBase object

`needs_fitting`(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need to be fit or whose fit methods do nothing.

**Returns** True.

`property parameters`(*self*)

Returns the parameters which were used to initialize the component.

`save`(*self*, *file_path*, *pickle_protocol=cloudpickle.DEFAULT_PROTOCOL*)

Saves component at file path.

**Parameters**

- **file_path** (*str*) – Location to save file.
- **pickle_protocol** (*int*) – The pickle data stream format.

**transform**(*self*, *X*, *y=None*)

Transforms data X by imputing missing values using specified timeseries-specific strategies. 'None' values are converted to np.nan before imputation and are treated as the same.

**Parameters**

- **X** (*pd.DataFrame*) – Data to transform.
- **y** (*pd.Series, optional*) – Optionally, target data to transform.

**Returns** Transformed X and y

**Return type** pd.DataFrame

**update_parameters**(*self*, *update_dict*, *reset_fit=True*)

Updates the parameter dictionary of the component.

**Parameters**

- **update_dict** (*dict*) – A dict of parameters to update.
- **reset_fit** (*bool, optional*) – If True, will set *_is_fitted* to False.

**class** evalml.pipelines.components.**TimeSeriesRegularizer**(*time_index=None*, *frequency_payload=None*, *window_length=4*, *threshold=0.4*, *random_seed=0*, ***kwargs*)

Transformer that regularizes an inconsistently spaced datetime column.

If X is passed in to fit/transform, the column *time_index* will be checked for an inferrable offset frequency. If the *time_index* column is perfectly inferrable then this Transformer will do nothing and return the original X and y.

If X does not have a perfectly inferrable frequency but one can be estimated, then X and y will be reformatted based on the estimated frequency for *time_index*. In the original X and y passed: - Missing datetime values will be added and will have their corresponding columns in X and y set to None. - Duplicate datetime values will be dropped. - Extra datetime values will be dropped. - If it can be determined that a duplicate or extra value is misaligned, then it will be repositioned to take the place of a missing value.

This Transformer should be used before the *TimeSeriesImputer* in order to impute the missing values that were added to X and y (if passed).

**Parameters**

- **time_index** (*string*) – Name of the column containing the datetime information used to order the data, required. Defaults to None.
- **frequency_payload** (*tuple*) – Payload returned from Woodwork's infer_frequency function where debug is True. Defaults to None.
- **window_length** (*int*) – The size of the rolling window over which inference is conducted to determine the prevalence of uninferrable frequencies.
- **5.** (*Lower values make this component more sensitive to recognizing numerous faulty datetime values. Defaults to*) –
- **threshold** (*float*) – The minimum percentage of windows that need to have been able to infer a frequency. Lower values make this component more
- **0.8.** (*sensitive to recognizing numerous faulty datetime values. Defaults to*) –
- **random_seed** (*int*) – Seed for the random number generator. This transformer performs the same regardless of the random seed provided.

- **0.** (*Defaults to*) –

**Raises** **ValueError** – if the frequency_payload parameter has not been passed a tuple

**Attributes**

| | |
|---|---|
| **hyper-parameter_ranges** | {} |
| **modifies_features** | True |
| **modifies_target** | True |
| **name** | Time Series Regularizer |
| **training_only** | True |

**Methods**

| | |
|---|---|
| *clone* | Constructs a new component with the same parameters and random state. |
| *default_parameters* | Returns the default parameters for this component. |
| *describe* | Describe a component and its parameters. |
| *fit* | Fits the TimeSeriesRegularizer. |
| *fit_transform* | Fits on X and transforms X. |
| *load* | Loads component at file path. |
| *needs_fitting* | Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances. |
| *parameters* | Returns the parameters which were used to initialize the component. |
| *save* | Saves component at file path. |
| *transform* | Regularizes a dataframe and target data to an inferrable offset frequency. |
| *update_parameters* | Updates the parameter dictionary of the component. |

**clone**(*self*)

Constructs a new component with the same parameters and random state.

**Returns** A new instance of this component with identical parameters and random state.

**default_parameters**(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

**Returns** Default parameters for this component.

**Return type** dict

**describe**(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

**Parameters**

- **print_name** (*bool, optional*) – whether to print name of component

- **return_dict** (*bool, optional*) – whether to return description as dictionary in the format {"name": name, "parameters": parameters}

**Returns** Returns dictionary if return_dict is True, else None.

**Return type** None or dict

**fit**(*self*, *X*, *y=None*)

Fits the TimeSeriesRegularizer.

**Parameters**

- **X** (*pd.DataFrame*) – The input training data of shape [n_samples, n_features].
- **y** (*pd.Series, optional*) – The target training data of length [n_samples].

**Returns** self

**Raises**

- **ValueError** – if self.time_index is None, if X and y have different lengths, if *time_index* in X does not have an offset frequency that can be estimated
- **TypeError** – if the *time_index* column is not of type Datetime
- **KeyError** – if the *time_index* column doesn't exist

**fit_transform**(*self*, *X*, *y=None*)

Fits on X and transforms X.

**Parameters**

- **X** (*pd.DataFrame*) – Data to fit and transform.
- **y** (*pd.Series*) – Target data.

**Returns** Transformed X.

**Return type** pd.DataFrame

**Raises** **MethodPropertyNotFoundError** – If transformer does not have a transform method or a component_obj that implements transform.

**static load**(*file_path*)

Loads component at file path.

**Parameters** **file_path** (*str*) – Location to load file.

**Returns** ComponentBase object

**needs_fitting**(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need to be fit or whose fit methods do nothing.

**Returns** True.

**property parameters**(*self*)

Returns the parameters which were used to initialize the component.

**save**(*self*, *file_path*, *pickle_protocol=cloudpickle.DEFAULT_PROTOCOL*)

Saves component at file path.

**Parameters**

- **file_path** (*str*) – Location to save file.

- **pickle_protocol** (*int*) – The pickle data stream format.

**transform**(*self*, *X*, *y=None*)

Regularizes a dataframe and target data to an inferrable offset frequency.

A 'clean' X and y (if y was passed in) are created based on an inferrable offset frequency and matching datetime values with the original X and y are imputed into the clean X and y. Datetime values identified as misaligned are shifted into their appropriate position.

**Parameters**

- **X** (*pd.DataFrame*) – The input training data of shape [n_samples, n_features].
- **y** (*pd.Series, optional*) – The target training data of length [n_samples].

**Returns** Data with an inferrable *time_index* offset frequency.

**Return type** (pd.DataFrame, pd.Series)

**update_parameters**(*self*, *update_dict*, *reset_fit=True*)

Updates the parameter dictionary of the component.

**Parameters**

- **update_dict** (*dict*) – A dict of parameters to update.
- **reset_fit** (*bool, optional*) – If True, will set *_is_fitted* to False.

**class** evalml.pipelines.components.**Transformer**(*parameters=None*, *component_obj=None*, *random_seed=0*, ***kwargs*)

A component that may or may not need fitting that transforms data. These components are used before an estimator.

To implement a new Transformer, define your own class which is a subclass of Transformer, including a name and a list of acceptable ranges for any parameters to be tuned during the automl search (hyperparameters). Define an *__init__* method which sets up any necessary state and objects. Make sure your *__init__* only uses standard keyword arguments and calls *super().__init__()* with a parameters dict. You may also override the *fit*, *transform*, *fit_transform* and other methods in this class if appropriate.

To see some examples, check out the definitions of any Transformer component.

**Parameters**

- **parameters** (*dict*) – Dictionary of parameters for the component. Defaults to None.
- **component_obj** (*obj*) – Third-party objects useful in component implementation. Defaults to None.
- **random_seed** (*int*) – Seed for the random number generator. Defaults to 0.

**Attributes**

| | |
|---|---|
| **modifies_features** | True |
| **modifies_target** | False |
| **training_only** | False |

**Methods**

| | |
|---|---|
| *clone* | Constructs a new component with the same parameters and random state. |
| *default_parameters* | Returns the default parameters for this component. |
| *describe* | Describe a component and its parameters. |
| *fit* | Fits component to data. |
| *fit_transform* | Fits on X and transforms X. |
| *load* | Loads component at file path. |
| *name* | Returns string name of this component. |
| *needs_fitting* | Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances. |
| *parameters* | Returns the parameters which were used to initialize the component. |
| *save* | Saves component at file path. |
| *transform* | Transforms data X. |
| *update_parameters* | Updates the parameter dictionary of the component. |

**clone**(*self*)

Constructs a new component with the same parameters and random state.

**Returns** A new instance of this component with identical parameters and random state.

**default_parameters**(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

**Returns** Default parameters for this component.

**Return type** dict

**describe**(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

**Parameters**

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description as dictionary in the format {"name": name, "parameters": parameters}

**Returns** Returns dictionary if return_dict is True, else None.

**Return type** None or dict

**fit**(*self*, *X*, *y=None*)

Fits component to data.

**Parameters**

- **X** (*pd.DataFrame*) – The input training data of shape [n_samples, n_features]
- **y** (*pd.Series, optional*) – The target training data of length [n_samples]

**Returns** self

**Raises** **MethodPropertyNotFoundError** – If component does not have a fit method or a component_obj that implements fit.

**fit_transform**(*self*, *X*, *y=None*)

Fits on X and transforms X.

**Parameters**

- **X** (*pd.DataFrame*) – Data to fit and transform.
- **y** (*pd.Series*) – Target data.

**Returns** Transformed X.

**Return type** pd.DataFrame

**Raises** **MethodPropertyNotFoundError** – If transformer does not have a transform method or a component_obj that implements transform.

**static load**(*file_path*)

Loads component at file path.

**Parameters** **file_path** (*str*) – Location to load file.

**Returns** ComponentBase object

**property name**(*cls*)

Returns string name of this component.

**needs_fitting**(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need to be fit or whose fit methods do nothing.

**Returns** True.

**property parameters**(*self*)

Returns the parameters which were used to initialize the component.

**save**(*self*, *file_path*, *pickle_protocol=cloudpickle.DEFAULT_PROTOCOL*)

Saves component at file path.

**Parameters**

- **file_path** (*str*) – Location to save file.
- **pickle_protocol** (*int*) – The pickle data stream format.

**abstract transform**(*self*, *X*, *y=None*)

Transforms data X.

**Parameters**

- **X** (*pd.DataFrame*) – Data to transform.
- **y** (*pd.Series, optional*) – Target data.

**Returns** Transformed X

**Return type** pd.DataFrame

**Raises** **MethodPropertyNotFoundError** – If transformer does not have a transform method or a component_obj that implements transform.

**update_parameters**(*self*, *update_dict*, *reset_fit=True*)

Updates the parameter dictionary of the component.

**Parameters**

- **update_dict** (*dict*) – A dict of parameters to update.
- **reset_fit** (*bool, optional*) – If True, will set *_is_fitted* to False.

**class** evalml.pipelines.components.**Undersampler**(*sampling_ratio=0.25*, *sampling_ratio_dict=None*, *min_samples=100*, *min_percentage=0.1*, *random_seed=0*, ***kwargs*)

Initializes an undersampling transformer to downsample the majority classes in the dataset.

This component is only run during training and not during predict.

**Parameters**

- **sampling_ratio** (*float*) – The smallest minority:majority ratio that is accepted as 'balanced'. For instance, a 1:4 ratio would be represented as 0.25, while a 1:1 ratio is 1.0. Must be between 0 and 1, inclusive. Defaults to 0.25.
- **sampling_ratio_dict** (*dict*) – A dictionary specifying the desired balanced ratio for each target value. For instance, in a binary case where class 1 is the minority, we could specify: *sampling_ratio_dict={0: 0.5, 1: 1}*, which means we would undersample class 0 to have twice the number of samples as class 1 (minority:majority ratio = 0.5), and don't sample class 1. Overrides sampling_ratio if provided. Defaults to None.
- **min_samples** (*int*) – The minimum number of samples that we must have for any class, pre or post sampling. If a class must be downsampled, it will not be downsampled past this value. To determine severe imbalance, the minority class must occur less often than this and must have a class ratio below min_percentage. Must be greater than 0. Defaults to 100.
- **min_percentage** (*float*) – The minimum percentage of the minimum class to total dataset that we tolerate, as long as it is above min_samples. If min_percentage and min_samples are not met, treat this as severely imbalanced, and we will not resample the data. Must be between 0 and 0.5, inclusive. Defaults to 0.1.
- **random_seed** (*int*) – The seed to use for random sampling. Defaults to 0.

**Raises**

- **ValueError** – If sampling_ratio is not in the range (0, 1].
- **ValueError** – If min_sample is not greater than 0.
- **ValueError** – If min_percentage is not between 0 and 0.5, inclusive.

**Attributes**

| | |
|---|---|
| **hyper-parameter_ranges** | {} |
| **modifies_features** | True |
| **modifies_target** | True |
| **name** | Undersampler |
| **training_only** | True |

**Methods**

| | |
|---|---|
| *clone* | Constructs a new component with the same parameters and random state. |
| *default_parameters* | Returns the default parameters for this component. |
| *describe* | Describe a component and its parameters. |
| *fit* | Fits the sampler to the data. |
| *fit_resample* | Resampling technique for this sampler. |
| *fit_transform* | Fit and transform data using the sampler component. |
| *load* | Loads component at file path. |
| *needs_fitting* | Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances. |
| *parameters* | Returns the parameters which were used to initialize the component. |
| *save* | Saves component at file path. |
| *transform* | Transforms the input data by sampling the data. |
| *update_parameters* | Updates the parameter dictionary of the component. |

**clone**(*self*)

Constructs a new component with the same parameters and random state.

**Returns** A new instance of this component with identical parameters and random state.

**default_parameters**(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

**Returns** Default parameters for this component.

**Return type** dict

**describe**(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

**Parameters**

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description as dictionary in the format {"name": name, "parameters": parameters}

**Returns** Returns dictionary if return_dict is True, else None.

**Return type** None or dict

**fit**(*self*, *X*, *y*)

Fits the sampler to the data.

**Parameters**

- **X** (*pd.DataFrame*) – Input features.
- **y** (*pd.Series*) – Target.

**Returns** self

**Raises** **ValueError** – If y is None.

**fit_resample**(*self*, *X*, *y*)

Resampling technique for this sampler.

**Parameters**

- **X** (*pd.DataFrame*) – Training data to fit and resample.
- **y** (*pd.Series*) – Training data targets to fit and resample.

**Returns** Indices to keep for training data.

**Return type** list

**fit_transform**(*self*, *X*, *y*)

Fit and transform data using the sampler component.

**Parameters**

- **X** (*pd.DataFrame*) – The input training data of shape [n_samples, n_features].
- **y** (*pd.Series, optional*) – The target training data of length [n_samples].

**Returns** Transformed data.

**Return type** (pd.DataFrame, pd.Series)

**static load**(*file_path*)

Loads component at file path.

**Parameters** **file_path** (*str*) – Location to load file.

**Returns** ComponentBase object

**needs_fitting**(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need to be fit or whose fit methods do nothing.

**Returns** True.

**property parameters**(*self*)

Returns the parameters which were used to initialize the component.

**save**(*self*, *file_path*, *pickle_protocol=cloudpickle.DEFAULT_PROTOCOL*)

Saves component at file path.

**Parameters**

- **file_path** (*str*) – Location to save file.
- **pickle_protocol** (*int*) – The pickle data stream format.

**transform**(*self*, *X*, *y=None*)

Transforms the input data by sampling the data.

**Parameters**

- **X** (*pd.DataFrame*) – Training features.
- **y** (*pd.Series*) – Target.

**Returns** Transformed features and target.

**Return type** pd.DataFrame, pd.Series

**update_parameters**(*self*, *update_dict*, *reset_fit=True*)

Updates the parameter dictionary of the component.

**Parameters**

- **update_dict** (*dict*) – A dict of parameters to update.
- **reset_fit** (*bool, optional*) – If True, will set *_is_fitted* to False.

**class** evalml.pipelines.components.**URLFeaturizer**(*random_seed=0*, ***kwargs*)

Transformer that can automatically extract features from URL.

**Parameters** **random_seed** (*int*) – Seed for the random number generator. Defaults to 0.

**Attributes**

| | |
|---|---|
| **hyper-parameter_ranges** | {} |
| **modifies_features** | True |
| **modifies_target** | False |
| **name** | URL Featurizer |
| **training_only** | False |

**Methods**

| | |
|---|---|
| *clone* | Constructs a new component with the same parameters and random state. |
| *default_parameters* | Returns the default parameters for this component. |
| *describe* | Describe a component and its parameters. |
| *fit* | Fits component to data. |
| *fit_transform* | Fits on X and transforms X. |
| *load* | Loads component at file path. |
| *needs_fitting* | Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances. |
| *parameters* | Returns the parameters which were used to initialize the component. |
| *save* | Saves component at file path. |
| *transform* | Transforms data X. |
| *update_parameters* | Updates the parameter dictionary of the component. |

**clone**(*self*)

Constructs a new component with the same parameters and random state.

**Returns** A new instance of this component with identical parameters and random state.

**default_parameters**(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

**Returns** Default parameters for this component.

**Return type** dict

**describe**(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

**Parameters**

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description as dictionary in the format {"name": name, "parameters": parameters}

**Returns** Returns dictionary if return_dict is True, else None.

**Return type** None or dict

**fit**(*self*, *X*, *y=None*)

Fits component to data.

**Parameters**

- **X** (*pd.DataFrame*) – The input training data of shape [n_samples, n_features]
- **y** (*pd.Series, optional*) – The target training data of length [n_samples]

**Returns** self

**Raises** **MethodPropertyNotFoundError** – If component does not have a fit method or a component_obj that implements fit.

**fit_transform**(*self*, *X*, *y=None*)

Fits on X and transforms X.

**Parameters**

- **X** (*pd.DataFrame*) – Data to fit and transform.
- **y** (*pd.Series*) – Target data.

**Returns** Transformed X.

**Return type** pd.DataFrame

**Raises** **MethodPropertyNotFoundError** – If transformer does not have a transform method or a component_obj that implements transform.

**static load**(*file_path*)

Loads component at file path.

**Parameters** **file_path** (*str*) – Location to load file.

**Returns** ComponentBase object

**needs_fitting**(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need to be fit or whose fit methods do nothing.

**Returns** True.

**property parameters**(*self*)

Returns the parameters which were used to initialize the component.

**save**(*self*, *file_path*, *pickle_protocol=cloudpickle.DEFAULT_PROTOCOL*)

Saves component at file path.

**Parameters**

- **file_path** (*str*) – Location to save file.
- **pickle_protocol** (*int*) – The pickle data stream format.

**transform**(*self*, *X*, *y=None*)

Transforms data X.

**Parameters**

- **X** (*pd.DataFrame*) – Data to transform.
- **y** (*pd.Series, optional*) – Target data.

**Returns** Transformed X

**Return type** pd.DataFrame

**Raises** **MethodPropertyNotFoundError** – If transformer does not have a transform method or a component_obj that implements transform.

**update_parameters**(*self*, *update_dict*, *reset_fit=True*)

Updates the parameter dictionary of the component.

**Parameters**

- **update_dict** (*dict*) – A dict of parameters to update.
- **reset_fit** (*bool, optional*) – If True, will set *_is_fitted* to False.

**class** evalml.pipelines.components.**VowpalWabbitBinaryClassifier**(*loss_function='logistic'*, *learning_rate=0.5*, *decay_learning_rate=1.0*, *power_t=0.5*, *passes=1*, *random_seed=0*, ***kwargs*)

Vowpal Wabbit Binary Classifier.

**Parameters**

- **loss_function** (*str*) – Specifies the loss function to use. One of {"squared", "classic", "hinge", "logistic", "quantile"}. Defaults to "logistic".
- **learning_rate** (*float*) – Boosting learning rate. Defaults to 0.5.
- **decay_learning_rate** (*float*) – Decay factor for learning_rate. Defaults to 1.0.
- **power_t** (*float*) – Power on learning rate decay. Defaults to 0.5.
- **passes** (*int*) – Number of training passes. Defaults to 1.
- **random_seed** (*int*) – Seed for the random number generator. Defaults to 0.

**Attributes**

| | |
|---|---|
| **hyper-parameter_ranges** | None |
| **model_family** | ModelFamily.VOWPAL_WABBIT |
| **modifies_features** | True |
| **modifies_target** | False |
| **name** | Vowpal Wabbit Binary Classifier |
| **supported_problem_types** | [ ProblemTypes.BINARY, ProblemTypes.TIME_SERIES_BINARY,] |
| **training_only** | False |

**Methods**

| | |
|---|---|
| *clone* | Constructs a new component with the same parameters and random state. |
| *default_parameters* | Returns the default parameters for this component. |
| *describe* | Describe a component and its parameters. |
| *feature_importance* | Feature importance for Vowpal Wabbit classifiers. This is not implemented. |
| *fit* | Fits estimator to data. |
| *get_prediction_intervals* | Find the prediction intervals using the fitted regressor. |
| *load* | Loads component at file path. |
| *needs_fitting* | Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances. |
| *parameters* | Returns the parameters which were used to initialize the component. |
| *predict* | Make predictions using selected features. |
| *predict_proba* | Make probability estimates for labels. |
| *save* | Saves component at file path. |
| *update_parameters* | Updates the parameter dictionary of the component. |

**clone**(*self*)

Constructs a new component with the same parameters and random state.

**Returns** A new instance of this component with identical parameters and random state.

**default_parameters**(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

**Returns** Default parameters for this component.

**Return type** dict

**describe**(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

**Parameters**

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description as dictionary in the format {"name": name, "parameters": parameters}

**Returns** Returns dictionary if return_dict is True, else None.

**Return type** None or dict

**property feature_importance**(*self*)

Feature importance for Vowpal Wabbit classifiers. This is not implemented.

**fit**(*self*, *X: pandas.DataFrame*, *y: Optional[pandas.Series] = None*)

Fits estimator to data.

**Parameters**

- **X** (*pd.DataFrame*) – The input training data of shape [n_samples, n_features].
- **y** (*pd.Series, optional*) – The target training data of length [n_samples].

**Returns** self

**get_prediction_intervals**(*self*, *X: pandas.DataFrame*, *y: Optional[pandas.Series] = None*, *coverage: List[float] = None*, *predictions: pandas.Series = None*) → Dict[str, pandas.Series]

Find the prediction intervals using the fitted regressor.

This function takes the predictions of the fitted estimator and calculates the rolling standard deviation across all predictions using a window size of 5. The lower and upper predictions are determined by taking the percent point (quantile) function of the lower tail probability at each bound multiplied by the rolling standard deviation.

**Parameters**

- **X** (*pd.DataFrame*) – Data of shape [n_samples, n_features].
- **y** (*pd.Series*) – Target data. Ignored.
- **coverage** (*list[float]*) – A list of floats between the values 0 and 1 that the upper and lower bounds of the prediction interval should be calculated for.
- **predictions** (*pd.Series*) – Optional list of predictions to use. If None, will generate predictions using *X*.

**Returns** Prediction intervals, keys are in the format {coverage}_lower or {coverage}_upper.

**Return type** dict

**Raises** **MethodPropertyNotFoundError** – If the estimator does not support Time Series Regression as a problem type.

**static load**(*file_path*)

Loads component at file path.

**Parameters** **file_path** (*str*) – Location to load file.

**Returns** ComponentBase object

**needs_fitting**(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need to be fit or whose fit methods do nothing.

**Returns** True.

**property parameters**(*self*)

Returns the parameters which were used to initialize the component.

**predict**(*self*, *X: pandas.DataFrame*) → pandas.Series

Make predictions using selected features.

**Parameters** **X** (*pd.DataFrame*) – Data of shape [n_samples, n_features].

**Returns** Predicted values.

**Return type** pd.Series

**Raises** **MethodPropertyNotFoundError** – If estimator does not have a predict method or a component_obj that implements predict.

**predict_proba**(*self*, *X: pandas.DataFrame*) → pandas.Series

Make probability estimates for labels.

**Parameters** **X** (*pd.DataFrame*) – Features.

**Returns** Probability estimates.

**Return type** pd.Series

**Raises** **MethodPropertyNotFoundError** – If estimator does not have a predict_proba method or a component_obj that implements predict_proba.

**save**(*self*, *file_path*, *pickle_protocol=cloudpickle.DEFAULT_PROTOCOL*)

Saves component at file path.

**Parameters**

- **file_path** (*str*) – Location to save file.
- **pickle_protocol** (*int*) – The pickle data stream format.

**update_parameters**(*self*, *update_dict*, *reset_fit=True*)

Updates the parameter dictionary of the component.

**Parameters**

- **update_dict** (*dict*) – A dict of parameters to update.
- **reset_fit** (*bool, optional*) – If True, will set *_is_fitted* to False.

**class** evalml.pipelines.components.**VowpalWabbitMulticlassClassifier**(*loss_function='logistic'*, *learning_rate=0.5*, *decay_learning_rate=1.0*, *power_t=0.5*, *passes=1*, *random_seed=0*, ***kwargs*)

Vowpal Wabbit Multiclass Classifier.

**Parameters**

- **loss_function** (*str*) – Specifies the loss function to use. One of {"squared", "classic", "hinge", "logistic", "quantile"}. Defaults to "logistic".
- **learning_rate** (*float*) – Boosting learning rate. Defaults to 0.5.
- **decay_learning_rate** (*float*) – Decay factor for learning_rate. Defaults to 1.0.
- **power_t** (*float*) – Power on learning rate decay. Defaults to 0.5.

- **random_seed** (*int*) – Seed for the random number generator. Defaults to 0.

**Attributes**

| | |
|---|---|
| **hyper-parameter_ranges** | None |
| **model_family** | ModelFamily.VOWPAL_WABBIT |
| **modifies_features** | True |
| **modifies_target** | False |
| **name** | Vowpal Wabbit Multiclass Classifier |
| **supported_problem_types** | [ ProblemTypes.MULTICLASS, ProblemTypes.TIME_SERIES_MULTICLASS,] |
| **training_only** | False |

**Methods**

| | |
|---|---|
| *clone* | Constructs a new component with the same parameters and random state. |
| *default_parameters* | Returns the default parameters for this component. |
| *describe* | Describe a component and its parameters. |
| *feature_importance* | Feature importance for Vowpal Wabbit classifiers. This is not implemented. |
| *fit* | Fits estimator to data. |
| *get_prediction_intervals* | Find the prediction intervals using the fitted regressor. |
| *load* | Loads component at file path. |
| *needs_fitting* | Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances. |
| *parameters* | Returns the parameters which were used to initialize the component. |
| *predict* | Make predictions using selected features. |
| *predict_proba* | Make probability estimates for labels. |
| *save* | Saves component at file path. |
| *update_parameters* | Updates the parameter dictionary of the component. |

**clone**(*self*)

Constructs a new component with the same parameters and random state.

**Returns** A new instance of this component with identical parameters and random state.

**default_parameters**(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

**Returns** Default parameters for this component.

**Return type** dict

**describe**(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

**Parameters**

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description as dictionary in the format {"name": name, "parameters": parameters}

**Returns** Returns dictionary if return_dict is True, else None.

**Return type** None or dict

**property feature_importance**(*self*)

Feature importance for Vowpal Wabbit classifiers. This is not implemented.

**fit**(*self*, *X: pandas.DataFrame*, *y: Optional[pandas.Series] = None*)

Fits estimator to data.

**Parameters**

- **X** (*pd.DataFrame*) – The input training data of shape [n_samples, n_features].
- **y** (*pd.Series, optional*) – The target training data of length [n_samples].

**Returns** self

**get_prediction_intervals**(*self*, *X: pandas.DataFrame*, *y: Optional[pandas.Series] = None*, *coverage: List[float] = None*, *predictions: pandas.Series = None*) → Dict[str, pandas.Series]

Find the prediction intervals using the fitted regressor.

This function takes the predictions of the fitted estimator and calculates the rolling standard deviation across all predictions using a window size of 5. The lower and upper predictions are determined by taking the percent point (quantile) function of the lower tail probability at each bound multiplied by the rolling standard deviation.

**Parameters**

- **X** (*pd.DataFrame*) – Data of shape [n_samples, n_features].
- **y** (*pd.Series*) – Target data. Ignored.
- **coverage** (*list[float]*) – A list of floats between the values 0 and 1 that the upper and lower bounds of the prediction interval should be calculated for.
- **predictions** (*pd.Series*) – Optional list of predictions to use. If None, will generate predictions using *X*.

**Returns** Prediction intervals, keys are in the format {coverage}_lower or {coverage}_upper.

**Return type** dict

**Raises** **MethodPropertyNotFoundError** – If the estimator does not support Time Series Regression as a problem type.

**static load**(*file_path*)

Loads component at file path.

**Parameters** **file_path** (*str*) – Location to load file.

**Returns** ComponentBase object

**needs_fitting**(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need to be fit or whose fit methods do nothing.

**Returns** True.

**property parameters**(*self*)

Returns the parameters which were used to initialize the component.

**predict**(*self*, *X: pandas.DataFrame*) → pandas.Series

Make predictions using selected features.

**Parameters** **X** (*pd.DataFrame*) – Data of shape [n_samples, n_features].

**Returns** Predicted values.

**Return type** pd.Series

**Raises** **MethodPropertyNotFoundError** – If estimator does not have a predict method or a component_obj that implements predict.

**predict_proba**(*self*, *X: pandas.DataFrame*) → pandas.Series

Make probability estimates for labels.

**Parameters** **X** (*pd.DataFrame*) – Features.

**Returns** Probability estimates.

**Return type** pd.Series

**Raises** **MethodPropertyNotFoundError** – If estimator does not have a predict_proba method or a component_obj that implements predict_proba.

**save**(*self*, *file_path*, *pickle_protocol=cloudpickle.DEFAULT_PROTOCOL*)

Saves component at file path.

**Parameters**

- **file_path** (*str*) – Location to save file.
- **pickle_protocol** (*int*) – The pickle data stream format.

**update_parameters**(*self*, *update_dict*, *reset_fit=True*)

Updates the parameter dictionary of the component.

**Parameters**

- **update_dict** (*dict*) – A dict of parameters to update.
- **reset_fit** (*bool, optional*) – If True, will set *_is_fitted* to False.

**class** evalml.pipelines.components.**VowpalWabbitRegressor**(*learning_rate=0.5*, *decay_learning_rate=1.0*, *power_t=0.5*, *passes=1*, *random_seed=0*, ***kwargs*)

Vowpal Wabbit Regressor.

**Parameters**

- **learning_rate** (*float*) – Boosting learning rate. Defaults to 0.5.
- **decay_learning_rate** (*float*) – Decay factor for learning_rate. Defaults to 1.0.
- **power_t** (*float*) – Power on learning rate decay. Defaults to 0.5.

- **passes** (*int*) – Number of training passes. Defaults to 1.
- **random_seed** (*int*) – Seed for the random number generator. Defaults to 0.

**Attributes**

| | |
|---|---|
| **hyper-parameter_ranges** | None |
| **model_family** | ModelFamily.VOWPAL_WABBIT |
| **modifies_features** | True |
| **modifies_target** | False |
| **name** | Vowpal Wabbit Regressor |
| **supported_problem_types** | [ ProblemTypes.REGRESSION, ProblemTypes.TIME_SERIES_REGRESSION,] |
| **training_only** | False |

**Methods**

| | |
|---|---|
| *clone* | Constructs a new component with the same parameters and random state. |
| *default_parameters* | Returns the default parameters for this component. |
| *describe* | Describe a component and its parameters. |
| *feature_importance* | Feature importance for Vowpal Wabbit regressor. |
| *fit* | Fits estimator to data. |
| *get_prediction_intervals* | Find the prediction intervals using the fitted regressor. |
| *load* | Loads component at file path. |
| *needs_fitting* | Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances. |
| *parameters* | Returns the parameters which were used to initialize the component. |
| *predict* | Make predictions using selected features. |
| *predict_proba* | Make probability estimates for labels. |
| *save* | Saves component at file path. |
| *update_parameters* | Updates the parameter dictionary of the component. |

**clone**(*self*)

Constructs a new component with the same parameters and random state.

**Returns** A new instance of this component with identical parameters and random state.

**default_parameters**(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

**Returns** Default parameters for this component.

**Return type** dict

**describe**(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

**Parameters**

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description as dictionary in the format {"name": name, "parameters": parameters}

**Returns** Returns dictionary if return_dict is True, else None.

**Return type** None or dict

**property feature_importance**(*self*)

Feature importance for Vowpal Wabbit regressor.

**fit**(*self*, *X: pandas.DataFrame*, *y: Optional[pandas.Series] = None*)

Fits estimator to data.

**Parameters**

- **X** (*pd.DataFrame*) – The input training data of shape [n_samples, n_features].
- **y** (*pd.Series, optional*) – The target training data of length [n_samples].

**Returns** self

**get_prediction_intervals**(*self*, *X: pandas.DataFrame*, *y: Optional[pandas.Series] = None*, *coverage: List[float] = None*, *predictions: pandas.Series = None*) → Dict[str, pandas.Series]

Find the prediction intervals using the fitted regressor.

This function takes the predictions of the fitted estimator and calculates the rolling standard deviation across all predictions using a window size of 5. The lower and upper predictions are determined by taking the percent point (quantile) function of the lower tail probability at each bound multiplied by the rolling standard deviation.

**Parameters**

- **X** (*pd.DataFrame*) – Data of shape [n_samples, n_features].
- **y** (*pd.Series*) – Target data. Ignored.
- **coverage** (*list[float]*) – A list of floats between the values 0 and 1 that the upper and lower bounds of the prediction interval should be calculated for.
- **predictions** (*pd.Series*) – Optional list of predictions to use. If None, will generate predictions using *X*.

**Returns** Prediction intervals, keys are in the format {coverage}_lower or {coverage}_upper.

**Return type** dict

**Raises** **MethodPropertyNotFoundError** – If the estimator does not support Time Series Regression as a problem type.

**static load**(*file_path*)

Loads component at file path.

**Parameters** **file_path** (*str*) – Location to load file.

**Returns** ComponentBase object

**needs_fitting**(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need to be fit or whose fit methods do nothing.

**Returns** True.

**property parameters**(*self*)

Returns the parameters which were used to initialize the component.

**predict**(*self*, *X: pandas.DataFrame*) → pandas.Series

Make predictions using selected features.

**Parameters** **X** (*pd.DataFrame*) – Data of shape [n_samples, n_features].

**Returns** Predicted values.

**Return type** pd.Series

**Raises** **MethodPropertyNotFoundError** – If estimator does not have a predict method or a component_obj that implements predict.

**predict_proba**(*self*, *X: pandas.DataFrame*) → pandas.Series

Make probability estimates for labels.

**Parameters** **X** (*pd.DataFrame*) – Features.

**Returns** Probability estimates.

**Return type** pd.Series

**Raises** **MethodPropertyNotFoundError** – If estimator does not have a predict_proba method or a component_obj that implements predict_proba.

**save**(*self*, *file_path*, *pickle_protocol=cloudpickle.DEFAULT_PROTOCOL*)

Saves component at file path.

**Parameters**

- **file_path** (*str*) – Location to save file.
- **pickle_protocol** (*int*) – The pickle data stream format.

**update_parameters**(*self*, *update_dict*, *reset_fit=True*)

Updates the parameter dictionary of the component.

**Parameters**

- **update_dict** (*dict*) – A dict of parameters to update.
- **reset_fit** (*bool, optional*) – If True, will set *_is_fitted* to False.

**class** evalml.pipelines.components.**XGBoostClassifier**(*eta=0.1*, *max_depth=6*, *min_child_weight=1*, *n_estimators=100*, *random_seed=0*, *eval_metric='logloss'*, *n_jobs=12*, ***kwargs*)

XGBoost Classifier.

**Parameters**

- **eta** (*float*) – Boosting learning rate. Defaults to 0.1.
- **max_depth** (*int*) – Maximum tree depth for base learners. Defaults to 6.

- **min_child_weight** (*float*) – Minimum sum of instance weight (hessian) needed in a child. Defaults to 1.0
- **n_estimators** (*int*) – Number of gradient boosted trees. Equivalent to number of boosting rounds. Defaults to 100.
- **random_seed** (*int*) – Seed for the random number generator. Defaults to 0.
- **n_jobs** (*int*) – Number of parallel threads used to run xgboost. Note that creating thread contention will significantly slow down the algorithm. Defaults to 12.

**Attributes**

| | |
|---|---|
| **hyper-parameter_ranges** | { "eta": Real(0.000001, 1), "max_depth": Integer(1, 10), "min_child_weight": Real(1, 10), "n_estimators": Integer(1, 1000),} |
| **model_family** | ModelFamily.XGBOOST |
| **modifies_features** | True |
| **modifies_target** | False |
| **name** | XGBoost Classifier |
| **SEED_MAX** | None |
| **SEED_MIN** | None |
| **supported_problem_types** | [ ProblemTypes.BINARY, ProblemTypes.MULTICLASS, ProblemTypes.TIME_SERIES_BINARY, ProblemTypes.TIME_SERIES_MULTICLASS,] |
| **training_only** | False |

**Methods**

| | |
|---|---|
| *clone* | Constructs a new component with the same parameters and random state. |
| *default_parameters* | Returns the default parameters for this component. |
| *describe* | Describe a component and its parameters. |
| *feature_importance* | Feature importance of fitted XGBoost classifier. |
| *fit* | Fits XGBoost classifier component to data. |
| *get_prediction_intervals* | Find the prediction intervals using the fitted regressor. |
| *load* | Loads component at file path. |
| *needs_fitting* | Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances. |
| *parameters* | Returns the parameters which were used to initialize the component. |
| *predict* | Make predictions using the fitted XGBoost classifier. |
| *predict_proba* | Make predictions using the fitted CatBoost classifier. |
| *save* | Saves component at file path. |
| *update_parameters* | Updates the parameter dictionary of the component. |

**clone**(*self*)

Constructs a new component with the same parameters and random state.

**Returns** A new instance of this component with identical parameters and random state.

**default_parameters**(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

**Returns** Default parameters for this component.

**Return type** dict

**describe**(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

**Parameters**

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description as dictionary in the format {"name": name, "parameters": parameters}

**Returns** Returns dictionary if return_dict is True, else None.

**Return type** None or dict

**property feature_importance**(*self*)

Feature importance of fitted XGBoost classifier.

**fit**(*self*, *X*, *y=None*)

Fits XGBoost classifier component to data.

**Parameters**

- **X** (*pd.DataFrame*) – The input training data of shape [n_samples, n_features].
- **y** (*pd.Series*) – The target training data of length [n_samples].

**Returns** self

**get_prediction_intervals**(*self*, *X: pandas.DataFrame*, *y: Optional[pandas.Series] = None*, *coverage: List[float] = None*, *predictions: pandas.Series = None*) → Dict[str, pandas.Series]

Find the prediction intervals using the fitted regressor.

This function takes the predictions of the fitted estimator and calculates the rolling standard deviation across all predictions using a window size of 5. The lower and upper predictions are determined by taking the percent point (quantile) function of the lower tail probability at each bound multiplied by the rolling standard deviation.

**Parameters**

- **X** (*pd.DataFrame*) – Data of shape [n_samples, n_features].
- **y** (*pd.Series*) – Target data. Ignored.
- **coverage** (*list[float]*) – A list of floats between the values 0 and 1 that the upper and lower bounds of the prediction interval should be calculated for.
- **predictions** (*pd.Series*) – Optional list of predictions to use. If None, will generate predictions using *X*.

**Returns** Prediction intervals, keys are in the format {coverage}_lower or {coverage}_upper.

**Return type** dict

**Raises** **MethodPropertyNotFoundError** – If the estimator does not support Time Series Regression as a problem type.

**static load**(*file_path*)

Loads component at file path.

**Parameters** **file_path** (*str*) – Location to load file.

**Returns** ComponentBase object

**needs_fitting**(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need to be fit or whose fit methods do nothing.

**Returns** True.

**property parameters**(*self*)

Returns the parameters which were used to initialize the component.

**predict**(*self*, *X*)

Make predictions using the fitted XGBoost classifier.

**Parameters** **X** (*pd.DataFrame*) – Data of shape [n_samples, n_features].

**Returns** Predicted values.

**Return type** pd.DataFrame

**predict_proba**(*self*, *X*)

Make predictions using the fitted CatBoost classifier.

**Parameters** **X** (*pd.DataFrame*) – Data of shape [n_samples, n_features].

**Returns** Predicted values.

**Return type** pd.DataFrame

**save**(*self*, *file_path*, *pickle_protocol=cloudpickle.DEFAULT_PROTOCOL*)

Saves component at file path.

**Parameters**

- **file_path** (*str*) – Location to save file.
- **pickle_protocol** (*int*) – The pickle data stream format.

**update_parameters**(*self*, *update_dict*, *reset_fit=True*)

Updates the parameter dictionary of the component.

**Parameters**

- **update_dict** (*dict*) – A dict of parameters to update.
- **reset_fit** (*bool, optional*) – If True, will set *_is_fitted* to False.

**class** evalml.pipelines.components.**XGBoostRegressor**(*eta: float = 0.1*, *max_depth: int = 6*, *min_child_weight: int = 1*, *n_estimators: int = 100*, *random_seed: Union[int, float] = 0*, *n_jobs: int = 12*, ***kwargs*)

XGBoost Regressor.

**Parameters**

- **eta** (*float*) – Boosting learning rate. Defaults to 0.1.
- **max_depth** (*int*) – Maximum tree depth for base learners. Defaults to 6.

- **min_child_weight** (*float*) – Minimum sum of instance weight (hessian) needed in a child. Defaults to 1.0
- **n_estimators** (*int*) – Number of gradient boosted trees. Equivalent to number of boosting rounds. Defaults to 100.
- **random_seed** (*int*) – Seed for the random number generator. Defaults to 0.
- **n_jobs** (*int*) – Number of parallel threads used to run xgboost. Note that creating thread contention will significantly slow down the algorithm. Defaults to 12.

**Attributes**

| | |
|---|---|
| **hyper-parameter_ranges** | { "eta": Real(0.000001, 1), "max_depth": Integer(1, 20), "min_child_weight": Real(1, 10), "n_estimators": Integer(1, 1000),} |
| **model_family** | ModelFamily.XGBOOST |
| **modifies_features** | True |
| **modifies_target** | False |
| **name** | XGBoost Regressor |
| **SEED_MAX** | None |
| **SEED_MIN** | None |
| **supported_problem_types** | [ ProblemTypes.REGRESSION, ProblemTypes.TIME_SERIES_REGRESSION,] |
| **training_only** | False |

**Methods**

| | |
|---|---|
| *clone* | Constructs a new component with the same parameters and random state. |
| *default_parameters* | Returns the default parameters for this component. |
| *describe* | Describe a component and its parameters. |
| *feature_importance* | Feature importance of fitted XGBoost regressor. |
| *fit* | Fits XGBoost regressor component to data. |
| *get_prediction_intervals* | Find the prediction intervals using the fitted XGBoostRegressor. |
| *load* | Loads component at file path. |
| *needs_fitting* | Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances. |
| *parameters* | Returns the parameters which were used to initialize the component. |
| *predict* | Make predictions using fitted XGBoost regressor. |
| *predict_proba* | Make probability estimates for labels. |
| *save* | Saves component at file path. |
| *update_parameters* | Updates the parameter dictionary of the component. |

**clone**(*self*)

Constructs a new component with the same parameters and random state.

**Returns** A new instance of this component with identical parameters and random state.

**default_parameters**(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

**Returns** Default parameters for this component.

**Return type** dict

**describe**(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

**Parameters**

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description as dictionary in the format {"name": name, "parameters": parameters}

**Returns** Returns dictionary if return_dict is True, else None.

**Return type** None or dict

**property feature_importance**(*self*) → pandas.Series

Feature importance of fitted XGBoost regressor.

**fit**(*self*, *X: pandas.DataFrame*, *y: Optional[pandas.Series] = None*)

Fits XGBoost regressor component to data.

**Parameters**

- **X** (*pd.DataFrame*) – The input training data of shape [n_samples, n_features].
- **y** (*pd.Series, optional*) – The target training data of length [n_samples].

**Returns** self

**get_prediction_intervals**(*self*, *X: pandas.DataFrame*, *y: Optional[pandas.Series] = None*, *coverage: List[float] = None*, *predictions: pandas.Series = None*) → Dict[str, pandas.Series]

Find the prediction intervals using the fitted XGBoostRegressor.

**Parameters**

- **X** (*pd.DataFrame*) – Data of shape [n_samples, n_features].
- **y** (*pd.Series*) – Target data. Ignored.
- **coverage** (*List[float]*) – A list of floats between the values 0 and 1 that the upper and lower bounds of the prediction interval should be calculated for.
- **predictions** (*pd.Series*) – Optional list of predictions to use. If None, will generate predictions using *X*.

**Returns** Prediction intervals, keys are in the format {coverage}_lower or {coverage}_upper.

**Return type** dict

**static load**(*file_path*)

Loads component at file path.

**Parameters** **file_path** (*str*) – Location to load file.

**Returns** ComponentBase object

**needs_fitting**(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need to be fit or whose fit methods do nothing.

**Returns** True.

**property parameters**(*self*)

Returns the parameters which were used to initialize the component.

**predict**(*self*, *X: pandas.DataFrame*) → pandas.Series

Make predictions using fitted XGBoost regressor.

**Parameters** **X** (*pd.DataFrame*) – Data of shape [n_samples, n_features].

**Returns** Predicted values.

**Return type** pd.Series

**predict_proba**(*self*, *X: pandas.DataFrame*) → pandas.Series

Make probability estimates for labels.

**Parameters** **X** (*pd.DataFrame*) – Features.

**Returns** Probability estimates.

**Return type** pd.Series

**Raises** **MethodPropertyNotFoundError** – If estimator does not have a predict_proba method or a component_obj that implements predict_proba.

**save**(*self*, *file_path*, *pickle_protocol=cloudpickle.DEFAULT_PROTOCOL*)

Saves component at file path.

**Parameters**

- **file_path** (*str*) – Location to save file.
- **pickle_protocol** (*int*) – The pickle data stream format.

**update_parameters**(*self*, *update_dict*, *reset_fit=True*)

Updates the parameter dictionary of the component.

**Parameters**

- **update_dict** (*dict*) – A dict of parameters to update.
- **reset_fit** (*bool, optional*) – If True, will set *_is_fitted* to False.

## Submodules

### binary_classification_pipeline

Pipeline subclass for all binary classification pipelines.

### Module Contents

### Classes Summary

| | |
|---|---|
| *BinaryClassificationPipeline* | Pipeline subclass for all binary classification pipelines. |

### Contents

**class** evalml.pipelines.binary_classification_pipeline.**BinaryClassificationPipeline**(*component_graph, parameters=None, custom_name=None, random_seed=0*)

Pipeline subclass for all binary classification pipelines.

**Parameters**

- **component_graph** (*ComponentGraph, list, dict*) – ComponentGraph instance, list of components in order, or dictionary of components. Accepts strings or ComponentBase subclasses in the list. Note that when duplicate components are specified in a list, the duplicate component names will be modified with the component's index in the list. For example, the component graph [Imputer, One Hot Encoder, Imputer, Logistic Regression Classifier] will have names ["Imputer", "One Hot Encoder", "Imputer_2", "Logistic Regression Classifier"]
- **parameters** (*dict*) – Dictionary with component names as keys and dictionary of that component's parameters as values. An empty dictionary or None implies using all default values for component parameters. Defaults to None.
- **custom_name** (*str*) – Custom name for the pipeline. Defaults to None.
- **random_seed** (*int*) – Seed for the random number generator. Defaults to 0.

### Example

```
>>> pipeline = BinaryClassificationPipeline(component_graph=["Simple Imputer",
↪"Logistic Regression Classifier"],
...                                         parameters={"Logistic Regression␣
↪Classifier": {"penalty": "elasticnet",
...                                                                          ␣
↪       "solver": "liblinear"}},
...                                         custom_name="My Binary Pipeline")
...
>>> assert pipeline.custom_name == "My Binary Pipeline"
>>> assert pipeline.component_graph.component_dict.keys() == {'Simple Imputer',
↪'Logistic Regression Classifier'}
```

The pipeline parameters will be chosen from the default parameters for every component, unless specific parameters were passed in as they were above.

```
>>> assert pipeline.parameters == {
...     'Simple Imputer': {'impute_strategy': 'most_frequent', 'fill_value': None},
...     'Logistic Regression Classifier': {'penalty': 'elasticnet',
...                                        'C': 1.0,
...                                        'n_jobs': -1,
...                                        'multi_class': 'auto',
...                                        'solver': 'liblinear'}}
```

**Attributes**

| | |
|---|---|
| **problem_type** | ProblemTypes.BINARY |

**Methods**

| | |
|---|---|
| *can_tune_threshold_with_objective* | Determine whether the threshold of a binary classification pipeline can be tuned. |
| *classes_* | Gets the class names for the pipeline. Will return None before pipeline is fit. |
| *clone* | Constructs a new pipeline with the same components, parameters, and random seed. |
| *create_objectives* | Create objective instances from a list of strings or objective classes. |
| *custom_name* | Custom name of the pipeline. |
| *describe* | Outputs pipeline details including component parameters. |
| *feature_importance* | Importance associated with each feature. Features dropped by the feature selection are excluded. |
| *fit* | Build a classification model. For string and categorical targets, classes are sorted by sorted(set(y)) and then are mapped to values between 0 and n_classes-1. |
| *fit_transform* | Fit and transform all components in the component graph, if all components are Transformers. |
| *get_component* | Returns component by name. |
| *get_hyperparameter_ranges* | Returns hyperparameter ranges from all components as a dictionary. |
| *graph* | Generate an image representing the pipeline graph. |
| *graph_dict* | Generates a dictionary with nodes consisting of the component names and parameters, and edges detailing component relationships. This dictionary is JSON serializable in most cases. |
| *graph_feature_importance* | Generate a bar graph of the pipeline's feature importance. |
| *inverse_transform* | Apply component inverse_transform methods to estimator predictions in reverse order. |
| *load* | Loads pipeline at file path. |
| *model_family* | Returns model family of this pipeline. |
| *name* | Name of the pipeline. |
| *new* | Constructs a new instance of the pipeline with the same component graph but with a different set of parameters. Not to be confused with python's __new__ method. |
| *optimize_threshold* | Optimize the pipeline threshold given the objective to use. Only used for binary problems with objectives whose thresholds can be tuned. |
| *parameters* | Parameter dictionary for this pipeline. |
| *predict* | Make predictions using selected features. |
| *predict_proba* | Make probability estimates for labels. Assumes that the column at index 1 represents the positive label case. |
| *save* | Saves pipeline at file path. |
| *score* | Evaluate model performance on objectives. |
| *summary* | A short summary of the pipeline structure, describing the list of components used. |
| *threshold* | Threshold used to make a prediction. Defaults to None. |
| *transform* | Transform the input. |
| *transform_all_but_final* | Transforms the data by applying all pre-processing components. |

**can_tune_threshold_with_objective**(*self*, *objective*)

Determine whether the threshold of a binary classification pipeline can be tuned.

**Parameters** **objective** (*ObjectiveBase*) – Primary AutoMLSearch objective.

**Returns** True if the pipeline threshold can be tuned.

**Return type** bool

**property classes_**(*self*)

Gets the class names for the pipeline. Will return None before pipeline is fit.

**clone**(*self*)

Constructs a new pipeline with the same components, parameters, and random seed.

**Returns** A new instance of this pipeline with identical components, parameters, and random seed.

**static create_objectives**(*objectives*)

Create objective instances from a list of strings or objective classes.

**property custom_name**(*self*)

Custom name of the pipeline.

**describe**(*self*, *return_dict=False*)

Outputs pipeline details including component parameters.

**Parameters** **return_dict** (*bool*) – If True, return dictionary of information about pipeline. Defaults to False.

**Returns** Dictionary of all component parameters if return_dict is True, else None.

**Return type** dict

**property feature_importance**(*self*)

Importance associated with each feature. Features dropped by the feature selection are excluded.

**Returns** Feature names and their corresponding importance

**Return type** pd.DataFrame

**fit**(*self*, *X*, *y*)

Build a classification model. For string and categorical targets, classes are sorted by sorted(set(y)) and then are mapped to values between 0 and n_classes-1.

**Parameters**

- **X** (*pd.DataFrame or np.ndarray*) – The input training data of shape [n_samples, n_features]
- **y** (*pd.Series, np.ndarray*) – The target training labels of length [n_samples]

**Returns** self

**Raises**

- **ValueError** – If the number of unique classes in y are not appropriate for the type of pipeline.
- **TypeError** – If the dtype is boolean but pd.NA exists in the series.
- **Exception** – For all other exceptions.

**fit_transform**(*self*, *X*, *y*)

Fit and transform all components in the component graph, if all components are Transformers.

**Parameters**

- **X** (*pd.DataFrame*) – Input features of shape [n_samples, n_features].
- **y** (*pd.Series*) – The target data of length [n_samples].

**Returns** Transformed output.

**Return type** pd.DataFrame

**Raises** **ValueError** – If final component is an Estimator.

**get_component**(*self*, *name*)

Returns component by name.

**Parameters** **name** (*str*) – Name of component.

**Returns** Component to return

**Return type** Component

**get_hyperparameter_ranges**(*self*, *custom_hyperparameters*)

Returns hyperparameter ranges from all components as a dictionary.

**Parameters** **custom_hyperparameters** (*dict*) – Custom hyperparameters for the pipeline.

**Returns** Dictionary of hyperparameter ranges for each component in the pipeline.

**Return type** dict

**graph**(*self*, *filepath=None*)

Generate an image representing the pipeline graph.

**Parameters** **filepath** (*str, optional*) – Path to where the graph should be saved. If set to None (as by default), the graph will not be saved.

**Returns** Graph object that can be directly displayed in Jupyter notebooks.

**Return type** graphviz.Digraph

**Raises**

- **RuntimeError** – If graphviz is not installed.
- **ValueError** – If path is not writeable.

**graph_dict**(*self*)

Generates a dictionary with nodes consisting of the component names and parameters, and edges detailing component relationships. This dictionary is JSON serializable in most cases.

x_edges specifies from which component feature data is being passed. y_edges specifies from which component target data is being passed. This can be used to build graphs across a variety of visualization tools. Template: {"Nodes": {"component_name": {"Name": class_name, "Parameters": parameters_attributes}, ...}}, "x_edges": [[from_component_name, to_component_name], [from_component_name, to_component_name], ...], "y_edges": [[from_component_name, to_component_name], [from_component_name, to_component_name], ...]}

**Returns** A dictionary representing the DAG structure.

**Return type** dag_dict (dict)

**graph_feature_importance**(*self*, *importance_threshold=0*)

Generate a bar graph of the pipeline's feature importance.

**Parameters** **importance_threshold** (*float, optional*) – If provided, graph features with a permutation importance whose absolute value is larger than importance_threshold. Defaults to zero.

**Returns** A bar graph showing features and their corresponding importance.

**Return type** plotly.Figure

**Raises** **ValueError** – If importance threshold is not valid.

**inverse_transform**(*self*, *y*)

Apply component inverse_transform methods to estimator predictions in reverse order.

Components that implement inverse_transform are PolynomialDecomposer, LogTransformer, LabelEncoder (tbd).

**Parameters** **y** (*pd.Series*) – Final component features.

**Returns** The inverse transform of the target.

**Return type** pd.Series

**static load**(*file_path*)

Loads pipeline at file path.

**Parameters** **file_path** (*str*) – Location to load file.

**Returns** PipelineBase object

**property model_family**(*self*)

Returns model family of this pipeline.

**property name**(*self*)

Name of the pipeline.

**new**(*self*, *parameters*, *random_seed=0*)

Constructs a new instance of the pipeline with the same component graph but with a different set of parameters. Not to be confused with python's __new__ method.

**Parameters**

- **parameters** (*dict*) – Dictionary with component names as keys and dictionary of that component's parameters as values. An empty dictionary or None implies using all default values for component parameters. Defaults to None.
- **random_seed** (*int*) – Seed for the random number generator. Defaults to 0.

**Returns** A new instance of this pipeline with identical components.

**optimize_threshold**(*self*, *X*, *y*, *y_pred_proba*, *objective*)

Optimize the pipeline threshold given the objective to use. Only used for binary problems with objectives whose thresholds can be tuned.

**Parameters**

- **X** (*pd.DataFrame*) – Input features.
- **y** (*pd.Series*) – Input target values.
- **y_pred_proba** (*pd.Series*) – The predicted probabilities of the target outputted by the pipeline.

- **objective** (*ObjectiveBase*) – The objective to threshold with. Must have a tunable threshold.

**Raises** **ValueError** – If objective is not optimizable.

**property parameters**(*self*)

Parameter dictionary for this pipeline.

**Returns** Dictionary of all component parameters.

**Return type** dict

**predict**(*self*, *X*, *objective=None*, *X_train=None*, *y_train=None*)

Make predictions using selected features.

Note: we cast y as ints first to address boolean values that may be returned from calculating predictions which we would not be able to otherwise transform if we originally had integer targets.

**Parameters**

- **X** (*pd.DataFrame*) – Data of shape [n_samples, n_features].
- **objective** (*Object or string*) – The objective to use to make predictions.
- **X_train** (*pd.DataFrame*) – Training data. Ignored. Only used for time series.
- **y_train** (*pd.Series*) – Training labels. Ignored. Only used for time series.

**Returns** Estimated labels.

**Return type** pd.Series

**predict_proba**(*self*, *X*, *X_train=None*, *y_train=None*)

Make probability estimates for labels. Assumes that the column at index 1 represents the positive label case.

**Parameters**

- **X** (*pd.DataFrame or np.ndarray*) – Data of shape [n_samples, n_features]
- **X_train** (*pd.DataFrame or np.ndarray or None*) – Training data. Ignored. Only used for time series.
- **y_train** (*pd.Series or None*) – Training labels. Ignored. Only used for time series.

**Returns** Probability estimates

**Return type** pd.Series

**save**(*self*, *file_path*, *pickle_protocol=cloudpickle.DEFAULT_PROTOCOL*)

Saves pipeline at file path.

**Parameters**

- **file_path** (*str*) – Location to save file.
- **pickle_protocol** (*int*) – The pickle data stream format.

**score**(*self*, *X*, *y*, *objectives*, *X_train=None*, *y_train=None*)

Evaluate model performance on objectives.

**Parameters**

- **X** (*pd.DataFrame*) – Data of shape [n_samples, n_features]
- **y** (*pd.Series*) – True labels of length [n_samples]

- **objectives** (*list*) – List of objectives to score
- **X_train** (*pd.DataFrame*) – Training data. Ignored. Only used for time series.
- **y_train** (*pd.Series*) – Training labels. Ignored. Only used for time series.

**Returns** Ordered dictionary of objective scores.

**Return type** dict

**property summary**(*self*)

A short summary of the pipeline structure, describing the list of components used.

Example: Logistic Regression Classifier w/ Simple Imputer + One Hot Encoder

**Returns** A string describing the pipeline structure.

**property threshold**(*self*)

Threshold used to make a prediction. Defaults to None.

**transform**(*self*, *X*, *y=None*)

Transform the input.

**Parameters**

- **X** (*pd.DataFrame, or np.ndarray*) – Data of shape [n_samples, n_features].
- **y** (*pd.Series*) – The target data of length [n_samples]. Defaults to None.

**Returns** Transformed output.

**Return type** pd.DataFrame

**transform_all_but_final**(*self*, *X*, *y=None*, *X_train=None*, *y_train=None*)

Transforms the data by applying all pre-processing components.

**Parameters**

- **X** (*pd.DataFrame*) – Input data to the pipeline to transform.
- **y** (*pd.Series or None*) – Targets corresponding to X. Optional.
- **X_train** (*pd.DataFrame or np.ndarray or None*) – Training data. Only used for time series.
- **y_train** (*pd.Series or None*) – Training labels. Only used for time series.

**Returns** New transformed features.

**Return type** pd.DataFrame

### binary_classification_pipeline_mixin

Binary classification pipeline mix-in class.

### Module Contents

### Classes Summary

| | |
|---|---|
| *BinaryClassificationPipelineMixin* | Binary classification pipeline mix-in class. |

### Contents

**class** evalml.pipelines.binary_classification_pipeline_mixin.**BinaryClassificationPipelineMixin**

Binary classification pipeline mix-in class.

**Methods**

| | |
|---|---|
| *optimize_threshold* | Optimize the pipeline threshold given the objective to use. Only used for binary problems with objectives whose thresholds can be tuned. |
| *threshold* | Threshold used to make a prediction. Defaults to None. |

**optimize_threshold**(*self*, *X*, *y*, *y_pred_proba*, *objective*)

Optimize the pipeline threshold given the objective to use. Only used for binary problems with objectives whose thresholds can be tuned.

**Parameters**

- **X** (*pd.DataFrame*) – Input features.
- **y** (*pd.Series*) – Input target values.
- **y_pred_proba** (*pd.Series*) – The predicted probabilities of the target outputted by the pipeline.
- **objective** (*ObjectiveBase*) – The objective to threshold with. Must have a tunable threshold.

**Raises** **ValueError** – If objective is not optimizable.

**property threshold**(*self*)

Threshold used to make a prediction. Defaults to None.

### classification_pipeline

Pipeline subclass for all classification pipelines.

## Module Contents

## Classes Summary

| | |
|---|---|
| *ClassificationPipeline* | Pipeline subclass for all classification pipelines. |

## Contents

**class** evalml.pipelines.classification_pipeline.**ClassificationPipeline**(*component_graph*, *parameters=None*, *custom_name=None*, *random_seed=0*)

Pipeline subclass for all classification pipelines.

**Parameters**

- **component_graph** (*list or dict*) – List of components in order. Accepts strings or ComponentBase subclasses in the list. Note that when duplicate components are specified in a list, the duplicate component names will be modified with the component's index in the list. For example, the component graph [Imputer, One Hot Encoder, Imputer, Logistic Regression Classifier] will have names ["Imputer", "One Hot Encoder", "Imputer_2", "Logistic Regression Classifier"]
- **parameters** (*dict*) – Dictionary with component names as keys and dictionary of that component's parameters as values. An empty dictionary or None implies using all default values for component parameters. Defaults to None.
- **custom_name** (*str*) – Custom name for the pipeline. Defaults to None.
- **random_seed** (*int*) – Seed for the random number generator. Defaults to 0.

**Attributes**

| | |
|---|---|
| **problem_type** | None |

**Methods**

| | |
|---|---|
| *can_tune_threshold_with_objective* | Determine whether the threshold of a binary classification pipeline can be tuned. |
| *classes_* | Gets the class names for the pipeline. Will return None before pipeline is fit. |
| *clone* | Constructs a new pipeline with the same components, parameters, and random seed. |
| *create_objectives* | Create objective instances from a list of strings or objective classes. |
| *custom_name* | Custom name of the pipeline. |
| *describe* | Outputs pipeline details including component parameters. |
| *feature_importance* | Importance associated with each feature. Features dropped by the feature selection are excluded. |
| *fit* | Build a classification model. For string and categorical targets, classes are sorted by sorted(set(y)) and then are mapped to values between 0 and n_classes-1. |
| *fit_transform* | Fit and transform all components in the component graph, if all components are Transformers. |
| *get_component* | Returns component by name. |
| *get_hyperparameter_ranges* | Returns hyperparameter ranges from all components as a dictionary. |
| *graph* | Generate an image representing the pipeline graph. |
| *graph_dict* | Generates a dictionary with nodes consisting of the component names and parameters, and edges detailing component relationships. This dictionary is JSON serializable in most cases. |
| *graph_feature_importance* | Generate a bar graph of the pipeline's feature importance. |
| *inverse_transform* | Apply component inverse_transform methods to estimator predictions in reverse order. |
| *load* | Loads pipeline at file path. |
| *model_family* | Returns model family of this pipeline. |
| *name* | Name of the pipeline. |
| *new* | Constructs a new instance of the pipeline with the same component graph but with a different set of parameters. Not to be confused with python's __new__ method. |
| *parameters* | Parameter dictionary for this pipeline. |
| *predict* | Make predictions using selected features. |
| *predict_proba* | Make probability estimates for labels. |
| *save* | Saves pipeline at file path. |
| *score* | Evaluate model performance on objectives. |
| *summary* | A short summary of the pipeline structure, describing the list of components used. |
| *transform* | Transform the input. |
| *transform_all_but_final* | Transforms the data by applying all pre-processing components. |

**can_tune_threshold_with_objective**(*self*, *objective*)

Determine whether the threshold of a binary classification pipeline can be tuned.

**Parameters** **objective** (*ObjectiveBase*) – Primary AutoMLSearch objective.

**Returns** True if the pipeline threshold can be tuned.

**Return type** bool

**property classes_**(*self*)

Gets the class names for the pipeline. Will return None before pipeline is fit.

**clone**(*self*)

Constructs a new pipeline with the same components, parameters, and random seed.

**Returns** A new instance of this pipeline with identical components, parameters, and random seed.

**static create_objectives**(*objectives*)

Create objective instances from a list of strings or objective classes.

**property custom_name**(*self*)

Custom name of the pipeline.

**describe**(*self*, *return_dict=False*)

Outputs pipeline details including component parameters.

**Parameters** **return_dict** (*bool*) – If True, return dictionary of information about pipeline. Defaults to False.

**Returns** Dictionary of all component parameters if return_dict is True, else None.

**Return type** dict

**property feature_importance**(*self*)

Importance associated with each feature. Features dropped by the feature selection are excluded.

**Returns** Feature names and their corresponding importance

**Return type** pd.DataFrame

**fit**(*self*, *X*, *y*)

Build a classification model. For string and categorical targets, classes are sorted by sorted(set(y)) and then are mapped to values between 0 and n_classes-1.

**Parameters**

- **X** (*pd.DataFrame or np.ndarray*) – The input training data of shape [n_samples, n_features]
- **y** (*pd.Series, np.ndarray*) – The target training labels of length [n_samples]

**Returns** self

**Raises**

- **ValueError** – If the number of unique classes in y are not appropriate for the type of pipeline.
- **TypeError** – If the dtype is boolean but pd.NA exists in the series.
- **Exception** – For all other exceptions.

**fit_transform**(*self*, *X*, *y*)

Fit and transform all components in the component graph, if all components are Transformers.

**Parameters**

- **X** (*pd.DataFrame*) – Input features of shape [n_samples, n_features].
- **y** (*pd.Series*) – The target data of length [n_samples].

**Returns** Transformed output.

**Return type** pd.DataFrame

**Raises** **ValueError** – If final component is an Estimator.

**get_component**(*self*, *name*)

Returns component by name.

**Parameters** **name** (*str*) – Name of component.

**Returns** Component to return

**Return type** Component

**get_hyperparameter_ranges**(*self*, *custom_hyperparameters*)

Returns hyperparameter ranges from all components as a dictionary.

**Parameters** **custom_hyperparameters** (*dict*) – Custom hyperparameters for the pipeline.

**Returns** Dictionary of hyperparameter ranges for each component in the pipeline.

**Return type** dict

**graph**(*self*, *filepath=None*)

Generate an image representing the pipeline graph.

**Parameters** **filepath** (*str, optional*) – Path to where the graph should be saved. If set to None (as by default), the graph will not be saved.

**Returns** Graph object that can be directly displayed in Jupyter notebooks.

**Return type** graphviz.Digraph

**Raises**

- **RuntimeError** – If graphviz is not installed.
- **ValueError** – If path is not writeable.

**graph_dict**(*self*)

Generates a dictionary with nodes consisting of the component names and parameters, and edges detailing component relationships. This dictionary is JSON serializable in most cases.

x_edges specifies from which component feature data is being passed. y_edges specifies from which component target data is being passed. This can be used to build graphs across a variety of visualization tools. Template: {"Nodes": {"component_name": {"Name": class_name, "Parameters": parameters_attributes}, ...}}, "x_edges": [[from_component_name, to_component_name], [from_component_name, to_component_name], ...], "y_edges": [[from_component_name, to_component_name], [from_component_name, to_component_name], ...]}

**Returns** A dictionary representing the DAG structure.

**Return type** dag_dict (dict)

**graph_feature_importance**(*self*, *importance_threshold=0*)

Generate a bar graph of the pipeline's feature importance.

**Parameters** **importance_threshold** (*float, optional*) – If provided, graph features with a permutation importance whose absolute value is larger than importance_threshold. Defaults to zero.

**Returns** A bar graph showing features and their corresponding importance.

**Return type** plotly.Figure

**Raises** `ValueError` – If importance threshold is not valid.

`inverse_transform`(*self*, *y*)

Apply component inverse_transform methods to estimator predictions in reverse order.

Components that implement inverse_transform are PolynomialDecomposer, LogTransformer, LabelEncoder (tbd).

**Parameters** `y` (*pd.Series*) – Final component features.

**Returns** The inverse transform of the target.

**Return type** pd.Series

`static load`(*file_path*)

Loads pipeline at file path.

**Parameters** `file_path` (*str*) – Location to load file.

**Returns** PipelineBase object

`property model_family`(*self*)

Returns model family of this pipeline.

`property name`(*self*)

Name of the pipeline.

`new`(*self*, *parameters*, *random_seed=0*)

Constructs a new instance of the pipeline with the same component graph but with a different set of parameters. Not to be confused with python's __new__ method.

**Parameters**

- `parameters` (*dict*) – Dictionary with component names as keys and dictionary of that component's parameters as values. An empty dictionary or None implies using all default values for component parameters. Defaults to None.
- `random_seed` (*int*) – Seed for the random number generator. Defaults to 0.

**Returns** A new instance of this pipeline with identical components.

`property parameters`(*self*)

Parameter dictionary for this pipeline.

**Returns** Dictionary of all component parameters.

**Return type** dict

`predict`(*self*, *X*, *objective=None*, *X_train=None*, *y_train=None*)

Make predictions using selected features.

Note: we cast y as ints first to address boolean values that may be returned from calculating predictions which we would not be able to otherwise transform if we originally had integer targets.

**Parameters**

- `X` (*pd.DataFrame*) – Data of shape [n_samples, n_features].
- `objective` (*Object or string*) – The objective to use to make predictions.
- `X_train` (*pd.DataFrame*) – Training data. Ignored. Only used for time series.

- **y_train** (*pd.Series*) – Training labels. Ignored. Only used for time series.

**Returns** Estimated labels.

**Return type** pd.Series

**predict_proba**(*self*, *X*, *X_train=None*, *y_train=None*)

Make probability estimates for labels.

**Parameters**

- **X** (*pd.DataFrame or np.ndarray*) – Data of shape [n_samples, n_features]
- **X_train** (*pd.DataFrame or np.ndarray or None*) – Training data. Ignored. Only used for time series.
- **y_train** (*pd.Series or None*) – Training labels. Ignored. Only used for time series.

**Returns** Probability estimates

**Return type** pd.DataFrame

**Raises** **ValueError** – If final component is not an estimator.

**save**(*self*, *file_path*, *pickle_protocol=cloudpickle.DEFAULT_PROTOCOL*)

Saves pipeline at file path.

**Parameters**

- **file_path** (*str*) – Location to save file.
- **pickle_protocol** (*int*) – The pickle data stream format.

**score**(*self*, *X*, *y*, *objectives*, *X_train=None*, *y_train=None*)

Evaluate model performance on objectives.

**Parameters**

- **X** (*pd.DataFrame*) – Data of shape [n_samples, n_features]
- **y** (*pd.Series*) – True labels of length [n_samples]
- **objectives** (*list*) – List of objectives to score
- **X_train** (*pd.DataFrame*) – Training data. Ignored. Only used for time series.
- **y_train** (*pd.Series*) – Training labels. Ignored. Only used for time series.

**Returns** Ordered dictionary of objective scores.

**Return type** dict

**property summary**(*self*)

A short summary of the pipeline structure, describing the list of components used.

Example: Logistic Regression Classifier w/ Simple Imputer + One Hot Encoder

**Returns** A string describing the pipeline structure.

**transform**(*self*, *X*, *y=None*)

Transform the input.

**Parameters**

- **X** (*pd.DataFrame, or np.ndarray*) – Data of shape [n_samples, n_features].
- **y** (*pd.Series*) – The target data of length [n_samples]. Defaults to None.

**Returns** Transformed output.

**Return type** pd.DataFrame

**transform_all_but_final**(*self*, *X*, *y=None*, *X_train=None*, *y_train=None*)

Transforms the data by applying all pre-processing components.

**Parameters**

- **X** (*pd.DataFrame*) – Input data to the pipeline to transform.
- **y** (*pd.Series or None*) – Targets corresponding to X. Optional.
- **X_train** (*pd.DataFrame or np.ndarray or None*) – Training data. Only used for time series.
- **y_train** (*pd.Series or None*) – Training labels. Only used for time series.

**Returns** New transformed features.

**Return type** pd.DataFrame

### component_graph

Component graph for a pipeline as a directed acyclic graph (DAG).

### Module Contents

### Classes Summary

| | |
|---|---|
| *ComponentGraph* | Component graph for a pipeline as a directed acyclic graph (DAG). |

### Attributes Summary

| |
|---|
| *logger* |

### Contents

**class** evalml.pipelines.component_graph.**ComponentGraph**(*component_dict=None*, *cached_data=None*, *random_seed=0*)

Component graph for a pipeline as a directed acyclic graph (DAG).

**Parameters**

- **component_dict** (*dict*) – A dictionary which specifies the components and edges between components that should be used to create the component graph. Defaults to None.
- **cached_data** (*dict*) – A dictionary of nested cached data. If the hashes and components are in this cache, we skip fitting for these components. Expected to be of format {hash1: {component_name: trained_component, ...}, hash2: {...}, ...}. Defaults to None.
- **random_seed** (*int*) – Seed for the random number generator. Defaults to 0.

### Examples

```
>>> component_dict = {'Imputer': ['Imputer', 'X', 'y'],
...                   'Logistic Regression': ['Logistic Regression Classifier',
↪'Imputer.x', 'y']}
>>> component_graph = ComponentGraph(component_dict)
>>> assert component_graph.compute_order == ['Imputer', 'Logistic Regression']
...
...
>>> component_dict = {'Imputer': ['Imputer', 'X', 'y'],
...                   'OHE': ['One Hot Encoder', 'Imputer.x', 'y'],
...                   'estimator_1': ['Random Forest Classifier', 'OHE.x', 'y'],
...                   'estimator_2': ['Decision Tree Classifier', 'OHE.x', 'y'],
...                   'final': ['Logistic Regression Classifier', 'estimator_1.x',
↪'estimator_2.x', 'y']}
>>> component_graph = ComponentGraph(component_dict)
```

The default parameters for every component in the component graph.

```
>>> assert component_graph.default_parameters == {
...     'Imputer': {'categorical_impute_strategy': 'most_frequent',
...                 'numeric_impute_strategy': 'mean',
...                 'boolean_impute_strategy': 'most_frequent',
...                 'categorical_fill_value': None,
...                 'numeric_fill_value': None,
...                 'boolean_fill_value': None},
...     'One Hot Encoder': {'top_n': 10,
...                         'features_to_encode': None,
...                         'categories': None,
...                         'drop': 'if_binary',
...                         'handle_unknown': 'ignore',
...                         'handle_missing': 'error'},
...     'Random Forest Classifier': {'n_estimators': 100,
...                                  'max_depth': 6,
...                                  'n_jobs': -1},
...     'Decision Tree Classifier': {'criterion': 'gini',
...                                  'max_features': 'auto',
...                                  'max_depth': 6,
...                                  'min_samples_split': 2,
...                                  'min_weight_fraction_leaf': 0.0},
...     'Logistic Regression Classifier': {'penalty': 'l2',
...                                        'C': 1.0,
...                                        'n_jobs': -1,
...                                        'multi_class': 'auto',
...                                        'solver': 'lbfgs'}}
```

**Methods**

| | |
|---|---|
| *compute_order* | The order that components will be computed or called in. |
| *default_parameters* | The default parameter dictionary for this pipeline. |
| *describe* | Outputs component graph details including component parameters. |
| *fit* | Fit each component in the graph. |
| *fit_and_transform_all_but_final* | Fit and transform all components save the final one, usually an estimator. |
| *fit_transform* | Fit and transform all components in the component graph, if all components are Transformers. |
| *generate_order* | Regenerated the topologically sorted order of the graph. |
| *get_component* | Retrieves a single component object from the graph. |
| *get_component_input_logical_types* | Get the logical types that are passed to the given component. |
| *get_estimators* | Gets a list of all the estimator components within this graph. |
| *get_inputs* | Retrieves all inputs for a given component. |
| *get_last_component* | Retrieves the component that is computed last in the graph, usually the final estimator. |
| *graph* | Generate an image representing the component graph. |
| *has_dfs* | Whether this component graph contains a DFSTransformer or not. |
| *instantiate* | Instantiates all uninstantiated components within the graph using the given parameters. An error will be raised if a component is already instantiated but the parameters dict contains arguments for that component. |
| *inverse_transform* | Apply component inverse_transform methods to estimator predictions in reverse order. |
| *last_component_input_logical_types* | Get the logical types that are passed to the last component in the pipeline. |
| *predict* | Make predictions using selected features. |
| *transform* | Transform the input using the component graph. |
| *transform_all_but_final* | Transform all components save the final one, and gathers the data from any number of parents to get all the information that should be fed to the final component. |

**property compute_order**(*self*)

The order that components will be computed or called in.

**property default_parameters**(*self*)

The default parameter dictionary for this pipeline.

**Returns** Dictionary of all component default parameters.

**Return type** dict

**describe**(*self*, *return_dict=False*)

Outputs component graph details including component parameters.

**Parameters** **return_dict** (*bool*) – If True, return dictionary of information about component

graph. Defaults to False.

**Returns** Dictionary of all component parameters if return_dict is True, else None

**Return type** dict

**Raises** `ValueError` – If the componentgraph is not instantiated

`fit`(*self*, *X*, *y*)

Fit each component in the graph.

**Parameters**

- **X** (*pd.DataFrame*) – The input training data of shape [n_samples, n_features].
- **y** (*pd.Series*) – The target training data of length [n_samples].

**Returns** self

`fit_and_transform_all_but_final`(*self*, *X*, *y*)

Fit and transform all components save the final one, usually an estimator.

**Parameters**

- **X** (*pd.DataFrame*) – The input training data of shape [n_samples, n_features].
- **y** (*pd.Series*) – The target training data of length [n_samples].

**Returns** Transformed features and target.

**Return type** Tuple (pd.DataFrame, pd.Series)

`fit_transform`(*self*, *X*, *y*)

Fit and transform all components in the component graph, if all components are Transformers.

**Parameters**

- **X** (*pd.DataFrame*) – Input features of shape [n_samples, n_features].
- **y** (*pd.Series*) – The target data of length [n_samples].

**Returns** Transformed output.

**Return type** pd.DataFrame

**Raises** `ValueError` – If final component is an Estimator.

`classmethod generate_order`(*cls*, *component_dict*)

Regenerated the topologically sorted order of the graph.

`get_component`(*self*, *component_name*)

Retrieves a single component object from the graph.

**Parameters** **component_name** (*str*) – Name of the component to retrieve

**Returns** ComponentBase object

**Raises** `ValueError` – If the component is not in the graph.

`get_component_input_logical_types`(*self*, *component_name*)

Get the logical types that are passed to the given component.

**Parameters** **component_name** (*str*) – Name of component in the graph

**Returns** Dict - Mapping feature name to logical type instance.

**Raises**

- **ValueError** – If the component is not in the graph.
- **ValueError** – If the component graph as not been fitted

**get_estimators**(*self*)

Gets a list of all the estimator components within this graph.

**Returns** All estimator objects within the graph.

**Return type** list

**Raises** **ValueError** – If the component graph is not yet instantiated.

**get_inputs**(*self*, *component_name*)

Retrieves all inputs for a given component.

**Parameters** **component_name** (*str*) – Name of the component to look up.

**Returns** List of inputs for the component to use.

**Return type** list[str]

**Raises** **ValueError** – If the component is not in the graph.

**get_last_component**(*self*)

Retrieves the component that is computed last in the graph, usually the final estimator.

**Returns** ComponentBase object

**Raises** **ValueError** – If the component graph has no edges.

**graph**(*self*, *name=None*, *graph_format=None*)

Generate an image representing the component graph.

**Parameters**

- **name** (*str*) – Name of the graph. Defaults to None.
- **graph_format** (*str*) – file format to save the graph in. Defaults to None.

**Returns** Graph object that can be directly displayed in Jupyter notebooks.

**Return type** graphviz.Digraph

**Raises** **RuntimeError** – If graphviz is not installed.

**property has_dfs**(*self*)

Whether this component graph contains a DFSTransformer or not.

**instantiate**(*self*, *parameters=None*)

Instantiates all uninstantiated components within the graph using the given parameters. An error will be raised if a component is already instantiated but the parameters dict contains arguments for that component.

**Parameters** **parameters** (*dict*) – Dictionary with component names as keys and dictionary of that component's parameters as values. An empty dictionary {} or None implies using all default values for component parameters. If a component in the component graph is already instantiated, it will not use any of its parameters defined in this dictionary. Defaults to None.

**Returns** self

**Raises** **ValueError** – If component graph is already instantiated or if a component errored while instantiating.

**inverse_transform**(*self*, *y*)

Apply component inverse_transform methods to estimator predictions in reverse order.

Components that implement inverse_transform are PolynomialDecomposer, LogTransformer, LabelEncoder (tbd).

**Parameters** **y** – (pd.Series): Final component features.

**Returns** The target with inverse transformation applied.

**Return type** pd.Series

**property last_component_input_logical_types**(*self*)

Get the logical types that are passed to the last component in the pipeline.

**Returns** Dict - Mapping feature name to logical type instance.

**Raises**

- **ValueError** – If the component is not in the graph.
- **ValueError** – If the component graph as not been fitted

**predict**(*self*, *X*)

Make predictions using selected features.

**Parameters** **X** (*pd.DataFrame*) – Input features of shape [n_samples, n_features].

**Returns** Predicted values.

**Return type** pd.Series

**Raises** **ValueError** – If final component is not an Estimator.

**transform**(*self*, *X*, *y=None*)

Transform the input using the component graph.

**Parameters**

- **X** (*pd.DataFrame*) – Input features of shape [n_samples, n_features].
- **y** (*pd.Series*) – The target data of length [n_samples]. Defaults to None.

**Returns** Transformed output.

**Return type** pd.DataFrame

**Raises** **ValueError** – If final component is not a Transformer.

**transform_all_but_final**(*self*, *X*, *y=None*)

Transform all components save the final one, and gathers the data from any number of parents to get all the information that should be fed to the final component.

**Parameters**

- **X** (*pd.DataFrame*) – Data of shape [n_samples, n_features].
- **y** (*pd.Series*) – The target training data of length [n_samples]. Defaults to None.

**Returns** Transformed values.

**Return type** pd.DataFrame

evalml.pipelines.component_graph.**logger**

### multiclass_classification_pipeline

Pipeline subclass for all multiclass classification pipelines.

### Module Contents

### Classes Summary

| | |
|---|---|
| *MulticlassClassificationPipeline* | Pipeline subclass for all multiclass classification pipelines. |

### Contents

**class** evalml.pipelines.multiclass_classification_pipeline.**MulticlassClassificationPipeline**(*component_grap*
*pa-*
*ram-*
*e-*
*ters=None*,
*cus-*
*tom_name=Non*
*ran-*
*dom_seed=0*)

Pipeline subclass for all multiclass classification pipelines.

**Parameters**

- **component_graph** (*ComponentGraph, list, dict*) – ComponentGraph instance, list of components in order, or dictionary of components. Accepts strings or ComponentBase subclasses in the list. Note that when duplicate components are specified in a list, the duplicate component names will be modified with the component's index in the list. For example, the component graph [Imputer, One Hot Encoder, Imputer, Logistic Regression Classifier] will have names ["Imputer", "One Hot Encoder", "Imputer_2", "Logistic Regression Classifier"]
- **parameters** (*dict*) – Dictionary with component names as keys and dictionary of that component's parameters as values. An empty dictionary or None implies using all default values for component parameters. Defaults to None.
- **custom_name** (*str*) – Custom name for the pipeline. Defaults to None.
- **random_seed** (*int*) – Seed for the random number generator. Defaults to 0.

### Example

```
>>> pipeline = MulticlassClassificationPipeline(component_graph=["Simple Imputer",
↪"Logistic Regression Classifier"],
...                                             parameters={"Logistic Regression␣
↪Classifier": {"penalty": "elasticnet",
...                                                                                ␣
↪            "solver": "liblinear"}},
...                                             custom_name="My Multiclass Pipeline
↪")
...
>>> assert pipeline.custom_name == "My Multiclass Pipeline"
>>> assert pipeline.component_graph.component_dict.keys() == {'Simple Imputer',
↪'Logistic Regression Classifier'}
```

The pipeline parameters will be chosen from the default parameters for every component, unless specific parameters were passed in as they were above.

```
>>> assert pipeline.parameters == {
...     'Simple Imputer': {'impute_strategy': 'most_frequent', 'fill_value': None},
...     'Logistic Regression Classifier': {'penalty': 'elasticnet',
...                                        'C': 1.0,
...                                        'n_jobs': -1,
...                                        'multi_class': 'auto',
...                                        'solver': 'liblinear'}}
```

#### Attributes

| | |
|---|---|
| **problem_type** | ProblemTypes.MULTICLASS |

#### Methods

| | |
|---|---|
| *can_tune_threshold_with_objective* | Determine whether the threshold of a binary classification pipeline can be tuned. |
| *classes_* | Gets the class names for the pipeline. Will return None before pipeline is fit. |
| *clone* | Constructs a new pipeline with the same components, parameters, and random seed. |
| *create_objectives* | Create objective instances from a list of strings or objective classes. |
| *custom_name* | Custom name of the pipeline. |
| *describe* | Outputs pipeline details including component parameters. |
| *feature_importance* | Importance associated with each feature. Features dropped by the feature selection are excluded. |
| *fit* | Build a classification model. For string and categorical targets, classes are sorted by sorted(set(y)) and then are mapped to values between 0 and n_classes-1. |
| *fit_transform* | Fit and transform all components in the component graph, if all components are Transformers. |
| *get_component* | Returns component by name. |
| *get_hyperparameter_ranges* | Returns hyperparameter ranges from all components as a dictionary. |
| *graph* | Generate an image representing the pipeline graph. |
| *graph_dict* | Generates a dictionary with nodes consisting of the component names and parameters, and edges detailing component relationships. This dictionary is JSON serializable in most cases. |
| *graph_feature_importance* | Generate a bar graph of the pipeline's feature importance. |
| *inverse_transform* | Apply component inverse_transform methods to estimator predictions in reverse order. |
| *load* | Loads pipeline at file path. |
| *model_family* | Returns model family of this pipeline. |
| *name* | Name of the pipeline. |
| *new* | Constructs a new instance of the pipeline with the same component graph but with a different set of parameters. Not to be confused with python's __new__ method. |
| *parameters* | Parameter dictionary for this pipeline. |
| *predict* | Make predictions using selected features. |
| *predict_proba* | Make probability estimates for labels. |
| *save* | Saves pipeline at file path. |
| *score* | Evaluate model performance on objectives. |
| *summary* | A short summary of the pipeline structure, describing the list of components used. |
| *transform* | Transform the input. |
| *transform_all_but_final* | Transforms the data by applying all pre-processing components. |

**can_tune_threshold_with_objective**(*self*, *objective*)

Determine whether the threshold of a binary classification pipeline can be tuned.

**Parameters** **objective** (*ObjectiveBase*) – Primary AutoMLSearch objective.

**Returns** True if the pipeline threshold can be tuned.

**Return type** bool

**property classes_**(*self*)

Gets the class names for the pipeline. Will return None before pipeline is fit.

**clone**(*self*)

Constructs a new pipeline with the same components, parameters, and random seed.

**Returns** A new instance of this pipeline with identical components, parameters, and random seed.

**static create_objectives**(*objectives*)

Create objective instances from a list of strings or objective classes.

**property custom_name**(*self*)

Custom name of the pipeline.

**describe**(*self*, *return_dict=False*)

Outputs pipeline details including component parameters.

**Parameters** **return_dict** (*bool*) – If True, return dictionary of information about pipeline. Defaults to False.

**Returns** Dictionary of all component parameters if return_dict is True, else None.

**Return type** dict

**property feature_importance**(*self*)

Importance associated with each feature. Features dropped by the feature selection are excluded.

**Returns** Feature names and their corresponding importance

**Return type** pd.DataFrame

**fit**(*self*, *X*, *y*)

Build a classification model. For string and categorical targets, classes are sorted by sorted(set(y)) and then are mapped to values between 0 and n_classes-1.

**Parameters**

- **X** (*pd.DataFrame or np.ndarray*) – The input training data of shape [n_samples, n_features]
- **y** (*pd.Series, np.ndarray*) – The target training labels of length [n_samples]

**Returns** self

**Raises**

- **ValueError** – If the number of unique classes in y are not appropriate for the type of pipeline.
- **TypeError** – If the dtype is boolean but pd.NA exists in the series.
- **Exception** – For all other exceptions.

**fit_transform**(*self*, *X*, *y*)

Fit and transform all components in the component graph, if all components are Transformers.

**Parameters**

- **X** (*pd.DataFrame*) – Input features of shape [n_samples, n_features].
- **y** (*pd.Series*) – The target data of length [n_samples].

**Returns** Transformed output.

**Return type** pd.DataFrame

**Raises** **ValueError** – If final component is an Estimator.

**get_component**(*self*, *name*)

Returns component by name.

**Parameters** **name** (*str*) – Name of component.

**Returns** Component to return

**Return type** Component

**get_hyperparameter_ranges**(*self*, *custom_hyperparameters*)

Returns hyperparameter ranges from all components as a dictionary.

**Parameters** **custom_hyperparameters** (*dict*) – Custom hyperparameters for the pipeline.

**Returns** Dictionary of hyperparameter ranges for each component in the pipeline.

**Return type** dict

**graph**(*self*, *filepath=None*)

Generate an image representing the pipeline graph.

**Parameters** **filepath** (*str, optional*) – Path to where the graph should be saved. If set to None (as by default), the graph will not be saved.

**Returns** Graph object that can be directly displayed in Jupyter notebooks.

**Return type** graphviz.Digraph

**Raises**

- **RuntimeError** – If graphviz is not installed.
- **ValueError** – If path is not writeable.

**graph_dict**(*self*)

Generates a dictionary with nodes consisting of the component names and parameters, and edges detailing component relationships. This dictionary is JSON serializable in most cases.

x_edges specifies from which component feature data is being passed. y_edges specifies from which component target data is being passed. This can be used to build graphs across a variety of visualization tools. Template: {"Nodes": {"component_name": {"Name": class_name, "Parameters": parameters_attributes}, ...}}, "x_edges": [[from_component_name, to_component_name], [from_component_name, to_component_name], ...], "y_edges": [[from_component_name, to_component_name], [from_component_name, to_component_name], ...]}

**Returns** A dictionary representing the DAG structure.

**Return type** dag_dict (dict)

**graph_feature_importance**(*self*, *importance_threshold=0*)

Generate a bar graph of the pipeline's feature importance.

**Parameters** **importance_threshold** (*float, optional*) – If provided, graph features with a permutation importance whose absolute value is larger than importance_threshold. Defaults to zero.

**Returns** A bar graph showing features and their corresponding importance.

**Return type** plotly.Figure

**Raises** `ValueError` – If importance threshold is not valid.

`inverse_transform`(*self*, *y*)

Apply component inverse_transform methods to estimator predictions in reverse order.

Components that implement inverse_transform are PolynomialDecomposer, LogTransformer, LabelEncoder (tbd).

**Parameters** `y` (*pd.Series*) – Final component features.

**Returns** The inverse transform of the target.

**Return type** pd.Series

`static load`(*file_path*)

Loads pipeline at file path.

**Parameters** `file_path` (*str*) – Location to load file.

**Returns** PipelineBase object

`property model_family`(*self*)

Returns model family of this pipeline.

`property name`(*self*)

Name of the pipeline.

`new`(*self*, *parameters*, *random_seed=0*)

Constructs a new instance of the pipeline with the same component graph but with a different set of parameters. Not to be confused with python's __new__ method.

**Parameters**

- `parameters` (*dict*) – Dictionary with component names as keys and dictionary of that component's parameters as values. An empty dictionary or None implies using all default values for component parameters. Defaults to None.
- `random_seed` (*int*) – Seed for the random number generator. Defaults to 0.

**Returns** A new instance of this pipeline with identical components.

`property parameters`(*self*)

Parameter dictionary for this pipeline.

**Returns** Dictionary of all component parameters.

**Return type** dict

`predict`(*self*, *X*, *objective=None*, *X_train=None*, *y_train=None*)

Make predictions using selected features.

Note: we cast y as ints first to address boolean values that may be returned from calculating predictions which we would not be able to otherwise transform if we originally had integer targets.

**Parameters**

- `X` (*pd.DataFrame*) – Data of shape [n_samples, n_features].
- `objective` (*Object or string*) – The objective to use to make predictions.
- `X_train` (*pd.DataFrame*) – Training data. Ignored. Only used for time series.

- **y_train** (*pd.Series*) – Training labels. Ignored. Only used for time series.

**Returns** Estimated labels.

**Return type** pd.Series

**predict_proba**(*self*, *X*, *X_train=None*, *y_train=None*)

Make probability estimates for labels.

**Parameters**

- **X** (*pd.DataFrame or np.ndarray*) – Data of shape [n_samples, n_features]
- **X_train** (*pd.DataFrame or np.ndarray or None*) – Training data. Ignored. Only used for time series.
- **y_train** (*pd.Series or None*) – Training labels. Ignored. Only used for time series.

**Returns** Probability estimates

**Return type** pd.DataFrame

**Raises** **ValueError** – If final component is not an estimator.

**save**(*self*, *file_path*, *pickle_protocol=cloudpickle.DEFAULT_PROTOCOL*)

Saves pipeline at file path.

**Parameters**

- **file_path** (*str*) – Location to save file.
- **pickle_protocol** (*int*) – The pickle data stream format.

**score**(*self*, *X*, *y*, *objectives*, *X_train=None*, *y_train=None*)

Evaluate model performance on objectives.

**Parameters**

- **X** (*pd.DataFrame*) – Data of shape [n_samples, n_features]
- **y** (*pd.Series*) – True labels of length [n_samples]
- **objectives** (*list*) – List of objectives to score
- **X_train** (*pd.DataFrame*) – Training data. Ignored. Only used for time series.
- **y_train** (*pd.Series*) – Training labels. Ignored. Only used for time series.

**Returns** Ordered dictionary of objective scores.

**Return type** dict

**property summary**(*self*)

A short summary of the pipeline structure, describing the list of components used.

Example: Logistic Regression Classifier w/ Simple Imputer + One Hot Encoder

**Returns** A string describing the pipeline structure.

**transform**(*self*, *X*, *y=None*)

Transform the input.

**Parameters**

- **X** (*pd.DataFrame, or np.ndarray*) – Data of shape [n_samples, n_features].
- **y** (*pd.Series*) – The target data of length [n_samples]. Defaults to None.

**Returns** Transformed output.

**Return type** pd.DataFrame

**transform_all_but_final**(*self*, *X*, *y=None*, *X_train=None*, *y_train=None*)

Transforms the data by applying all pre-processing components.

**Parameters**

- **X** (*pd.DataFrame*) – Input data to the pipeline to transform.
- **y** (*pd.Series or None*) – Targets corresponding to X. Optional.
- **X_train** (*pd.DataFrame or np.ndarray or None*) – Training data. Only used for time series.
- **y_train** (*pd.Series or None*) – Training labels. Only used for time series.

**Returns** New transformed features.

**Return type** pd.DataFrame

### pipeline_base

Base machine learning pipeline class.

### Module Contents

### Classes Summary

| | |
|---|---|
| *PipelineBase* | Machine learning pipeline. |

### Attributes Summary

| |
|---|
| *logger* |

### Contents

evalml.pipelines.pipeline_base.**logger**

**class** evalml.pipelines.pipeline_base.**PipelineBase**(*component_graph*, *parameters=None*, *custom_name=None*, *random_seed=0*)

Machine learning pipeline.

**Parameters**

- **component_graph** (*ComponentGraph, list, dict*) – ComponentGraph instance, list of components in order, or dictionary of components. Accepts strings or ComponentBase subclasses in the list. Note that when duplicate components are specified in a list, the duplicate component names will be modified with the component's index in the list. For example, the component graph [Imputer, One Hot Encoder, Imputer, Logistic Regression Classifier]

will have names ["Imputer", "One Hot Encoder", "Imputer_2", "Logistic Regression Classifier"].

- **parameters** (*dict*) – Dictionary with component names as keys and dictionary of that component's parameters as values. An empty dictionary or None implies using all default values for component parameters. Defaults to None.
- **custom_name** (*str*) – Custom name for the pipeline. Defaults to None.
- **random_seed** (*int*) – Seed for the random number generator. Defaults to 0.

**Attributes**

| | |
|---|---|
| **problem_type** | None |

**Methods**

| | |
|---|---|
| *can_tune_threshold_with_objective* | Determine whether the threshold of a binary classification pipeline can be tuned. |
| *clone* | Constructs a new pipeline with the same components, parameters, and random seed. |
| *create_objectives* | Create objective instances from a list of strings or objective classes. |
| *custom_name* | Custom name of the pipeline. |
| *describe* | Outputs pipeline details including component parameters. |
| *feature_importance* | Importance associated with each feature. Features dropped by the feature selection are excluded. |
| *fit* | Build a model. |
| *fit_transform* | Fit and transform all components in the component graph, if all components are Transformers. |
| *get_component* | Returns component by name. |
| *get_hyperparameter_ranges* | Returns hyperparameter ranges from all components as a dictionary. |
| *graph* | Generate an image representing the pipeline graph. |
| *graph_dict* | Generates a dictionary with nodes consisting of the component names and parameters, and edges detailing component relationships. This dictionary is JSON serializable in most cases. |
| *graph_feature_importance* | Generate a bar graph of the pipeline's feature importance. |
| *inverse_transform* | Apply component inverse_transform methods to estimator predictions in reverse order. |
| *load* | Loads pipeline at file path. |
| *model_family* | Returns model family of this pipeline. |
| *name* | Name of the pipeline. |
| *new* | Constructs a new instance of the pipeline with the same component graph but with a different set of parameters. Not to be confused with python's __new__ method. |
| *parameters* | Parameter dictionary for this pipeline. |
| *predict* | Make predictions using selected features. |
| *save* | Saves pipeline at file path. |
| *score* | Evaluate model performance on current and additional objectives. |
| *summary* | A short summary of the pipeline structure, describing the list of components used. |
| *transform* | Transform the input. |
| *transform_all_but_final* | Transforms the data by applying all pre-processing components. |

**can_tune_threshold_with_objective**(*self*, *objective*)

Determine whether the threshold of a binary classification pipeline can be tuned.

**Parameters** **objective** (*ObjectiveBase*) – Primary AutoMLSearch objective.

**Returns** True if the pipeline threshold can be tuned.

**Return type** bool

**clone**(*self*)

Constructs a new pipeline with the same components, parameters, and random seed.

**Returns** A new instance of this pipeline with identical components, parameters, and random seed.

**static create_objectives**(*objectives*)

Create objective instances from a list of strings or objective classes.

**property custom_name**(*self*)

Custom name of the pipeline.

**describe**(*self*, *return_dict=False*)

Outputs pipeline details including component parameters.

**Parameters** **return_dict** (*bool*) – If True, return dictionary of information about pipeline. Defaults to False.

**Returns** Dictionary of all component parameters if return_dict is True, else None.

**Return type** dict

**property feature_importance**(*self*)

Importance associated with each feature. Features dropped by the feature selection are excluded.

**Returns** Feature names and their corresponding importance

**Return type** pd.DataFrame

**abstract fit**(*self*, *X*, *y*)

Build a model.

**Parameters**

- **X** (*pd.DataFrame or np.ndarray*) – The input training data of shape [n_samples, n_features].
- **y** (*pd.Series, np.ndarray*) – The target training data of length [n_samples].

**Returns** self

**fit_transform**(*self*, *X*, *y*)

Fit and transform all components in the component graph, if all components are Transformers.

**Parameters**

- **X** (*pd.DataFrame*) – Input features of shape [n_samples, n_features].
- **y** (*pd.Series*) – The target data of length [n_samples].

**Returns** Transformed output.

**Return type** pd.DataFrame

**Raises** **ValueError** – If final component is an Estimator.

**get_component**(*self*, *name*)

Returns component by name.

**Parameters** **name** (*str*) – Name of component.

**Returns** Component to return

**Return type** Component

**get_hyperparameter_ranges**(*self*, *custom_hyperparameters*)

Returns hyperparameter ranges from all components as a dictionary.

**Parameters** **custom_hyperparameters** (*dict*) – Custom hyperparameters for the pipeline.

**Returns** Dictionary of hyperparameter ranges for each component in the pipeline.

**Return type** dict

**graph**(*self*, *filepath=None*)

Generate an image representing the pipeline graph.

**Parameters** **filepath** (*str, optional*) – Path to where the graph should be saved. If set to None (as by default), the graph will not be saved.

**Returns** Graph object that can be directly displayed in Jupyter notebooks.

**Return type** graphviz.Digraph

**Raises**

- **RuntimeError** – If graphviz is not installed.
- **ValueError** – If path is not writeable.

**graph_dict**(*self*)

Generates a dictionary with nodes consisting of the component names and parameters, and edges detailing component relationships. This dictionary is JSON serializable in most cases.

x_edges specifies from which component feature data is being passed. y_edges specifies from which component target data is being passed. This can be used to build graphs across a variety of visualization tools. Template: {"Nodes": {"component_name": {"Name": class_name, "Parameters": parameters_attributes}, ...}}, "x_edges": [[from_component_name, to_component_name], [from_component_name, to_component_name], ...], "y_edges": [[from_component_name, to_component_name], [from_component_name, to_component_name], ...]}

**Returns** A dictionary representing the DAG structure.

**Return type** dag_dict (dict)

**graph_feature_importance**(*self*, *importance_threshold=0*)

Generate a bar graph of the pipeline's feature importance.

**Parameters** **importance_threshold** (*float, optional*) – If provided, graph features with a permutation importance whose absolute value is larger than importance_threshold. Defaults to zero.

**Returns** A bar graph showing features and their corresponding importance.

**Return type** plotly.Figure

**Raises** **ValueError** – If importance threshold is not valid.

**inverse_transform**(*self*, *y*)

Apply component inverse_transform methods to estimator predictions in reverse order.

Components that implement inverse_transform are PolynomialDecomposer, LogTransformer, LabelEncoder (tbd).

**Parameters** **y** (*pd.Series*) – Final component features.

**Returns** The inverse transform of the target.

**Return type** pd.Series

**static load**(*file_path*)

Loads pipeline at file path.

**Parameters** **file_path** (*str*) – Location to load file.

**Returns** PipelineBase object

**property model_family**(*self*)

Returns model family of this pipeline.

**property name**(*self*)

Name of the pipeline.

**new**(*self*, *parameters*, *random_seed=0*)

Constructs a new instance of the pipeline with the same component graph but with a different set of parameters. Not to be confused with python's __new__ method.

**Parameters**

- **parameters** (*dict*) – Dictionary with component names as keys and dictionary of that component's parameters as values. An empty dictionary or None implies using all default values for component parameters. Defaults to None.
- **random_seed** (*int*) – Seed for the random number generator. Defaults to 0.

**Returns** A new instance of this pipeline with identical components.

**property parameters**(*self*)

Parameter dictionary for this pipeline.

**Returns** Dictionary of all component parameters.

**Return type** dict

**predict**(*self*, *X*, *objective=None*, *X_train=None*, *y_train=None*)

Make predictions using selected features.

**Parameters**

- **X** (*pd.DataFrame, or np.ndarray*) – Data of shape [n_samples, n_features].
- **objective** (*Object or string*) – The objective to use to make predictions.
- **X_train** (*pd.DataFrame or np.ndarray or None*) – Training data. Ignored. Only used for time series.
- **y_train** (*pd.Series or None*) – Training labels. Ignored. Only used for time series.

**Returns** Predicted values.

**Return type** pd.Series

**save**(*self*, *file_path*, *pickle_protocol=cloudpickle.DEFAULT_PROTOCOL*)

Saves pipeline at file path.

**Parameters**

- **file_path** (*str*) – Location to save file.
- **pickle_protocol** (*int*) – The pickle data stream format.

**abstract score**(*self*, *X*, *y*, *objectives*, *X_train=None*, *y_train=None*)

Evaluate model performance on current and additional objectives.

**Parameters**

- **X** (*pd.DataFrame or np.ndarray*) – Data of shape [n_samples, n_features].
- **y** (*pd.Series, np.ndarray*) – True labels of length [n_samples].
- **objectives** (*list*) – Non-empty list of objectives to score on.
- **X_train** (*pd.DataFrame or np.ndarray or None*) – Training data. Ignored. Only used for time series.
- **y_train** (*pd.Series or None*) – Training labels. Ignored. Only used for time series.

**Returns** Ordered dictionary of objective scores.

**Return type** dict

**property summary**(*self*)

A short summary of the pipeline structure, describing the list of components used.

Example: Logistic Regression Classifier w/ Simple Imputer + One Hot Encoder

**Returns** A string describing the pipeline structure.

**transform**(*self*, *X*, *y=None*)

Transform the input.

**Parameters**

- **X** (*pd.DataFrame, or np.ndarray*) – Data of shape [n_samples, n_features].
- **y** (*pd.Series*) – The target data of length [n_samples]. Defaults to None.

**Returns** Transformed output.

**Return type** pd.DataFrame

**transform_all_but_final**(*self*, *X*, *y=None*, *X_train=None*, *y_train=None*)

Transforms the data by applying all pre-processing components.

**Parameters**

- **X** (*pd.DataFrame*) – Input data to the pipeline to transform.
- **y** (*pd.Series or None*) – Targets corresponding to X. Optional.
- **X_train** (*pd.DataFrame or np.ndarray or None*) – Training data. Only used for time series.
- **y_train** (*pd.Series or None*) – Training labels. Only used for time series.

**Returns** New transformed features.

**Return type** pd.DataFrame

### pipeline_meta

Metaclass that overrides creating a new pipeline by wrapping methods with validators and setters.

## Module Contents

## Classes Summary

| | |
|---|---|
| *PipelineBaseMeta* | Metaclass that overrides creating a new pipeline by wrapping methods with validators and setters. |

## Contents

**class** evalml.pipelines.pipeline_meta.**PipelineBaseMeta**

Metaclass that overrides creating a new pipeline by wrapping methods with validators and setters.

**Attributes**

| | |
|---|---|
| **FIT_METHODS** | ['fit', 'fit_transform'] |
| **METHODS_TO_CHECK** | ['predict', 'predict_proba', 'transform', 'inverse_transform', 'get_trend_dataframe'] |
| **PROPERTIES_TO_CHECK** | ['feature_importance'] |

**Methods**

| | |
|---|---|
| *check_for_fit* | *check_for_fit* wraps a method that validates if *self._is_fitted* is *True*. |
| *register* | Register a virtual subclass of an ABC. |
| *set_fit* | Wrapper for the fit method. |

**classmethod check_for_fit**(*cls*, *method*)

*check_for_fit* wraps a method that validates if *self._is_fitted* is *True*.

**Parameters** **method** (*callable*) – Method to wrap.

**Returns** The wrapped method.

**Raises** **PipelineNotYetFittedError** – If pipeline is not yet fitted.

**register**(*cls*, *subclass*)

Register a virtual subclass of an ABC.

Returns the subclass, to allow usage as a class decorator.

**classmethod set_fit**(*cls*, *method*)

Wrapper for the fit method.

### regression_pipeline

Pipeline subclass for all regression pipelines.

### Module Contents

### Classes Summary

| | |
|---|---|
| *RegressionPipeline* | Pipeline subclass for all regression pipelines. |

### Contents

**class** evalml.pipelines.regression_pipeline.**RegressionPipeline**(*component_graph*, *parameters=None*, *custom_name=None*, *random_seed=0*)

Pipeline subclass for all regression pipelines.

**Parameters**

- **component_graph** (*ComponentGraph, list, dict*) – ComponentGraph instance, list of components in order, or dictionary of components. Accepts strings or ComponentBase subclasses in the list. Note that when duplicate components are specified in a list, the duplicate component names will be modified with the component's index in the list. For example, the component graph [Imputer, One Hot Encoder, Imputer, Logistic Regression Classifier] will have names ["Imputer", "One Hot Encoder", "Imputer_2", "Logistic Regression Classifier"]
- **parameters** (*dict*) – Dictionary with component names as keys and dictionary of that component's parameters as values. An empty dictionary or None implies using all default values for component parameters. Defaults to None.
- **custom_name** (*str*) – Custom name for the pipeline. Defaults to None.
- **random_seed** (*int*) – Seed for the random number generator. Defaults to 0.

### Example

```
>>> pipeline = RegressionPipeline(component_graph=["Simple Imputer", "Linear␣
↪Regressor"],
...                               parameters={"Simple Imputer": {"impute_strategy":
↪"mean"}},
...                               custom_name="My Regression Pipeline")
...
>>> assert pipeline.custom_name == "My Regression Pipeline"
>>> assert pipeline.component_graph.component_dict.keys() == {'Simple Imputer',
↪'Linear Regressor'}
```

The pipeline parameters will be chosen from the default parameters for every component, unless specific parameters were passed in as they were above.

```
>>> assert pipeline.parameters == {
...     'Simple Imputer': {'impute_strategy': 'mean', 'fill_value': None},
...     'Linear Regressor': {'fit_intercept': True, 'n_jobs': -1}}
```

**Attributes**

| | |
|---|---|
| **problem_type** | ProblemTypes.REGRESSION |

**Methods**

| | |
|---|---|
| *can_tune_threshold_with_objective* | Determine whether the threshold of a binary classification pipeline can be tuned. |
| *clone* | Constructs a new pipeline with the same components, parameters, and random seed. |
| *create_objectives* | Create objective instances from a list of strings or objective classes. |
| *custom_name* | Custom name of the pipeline. |
| *describe* | Outputs pipeline details including component parameters. |
| *feature_importance* | Importance associated with each feature. Features dropped by the feature selection are excluded. |
| *fit* | Build a regression model. |
| *fit_transform* | Fit and transform all components in the component graph, if all components are Transformers. |
| *get_component* | Returns component by name. |
| *get_hyperparameter_ranges* | Returns hyperparameter ranges from all components as a dictionary. |
| *graph* | Generate an image representing the pipeline graph. |
| *graph_dict* | Generates a dictionary with nodes consisting of the component names and parameters, and edges detailing component relationships. This dictionary is JSON serializable in most cases. |
| *graph_feature_importance* | Generate a bar graph of the pipeline's feature importance. |
| *inverse_transform* | Apply component inverse_transform methods to estimator predictions in reverse order. |
| *load* | Loads pipeline at file path. |
| *model_family* | Returns model family of this pipeline. |
| *name* | Name of the pipeline. |
| *new* | Constructs a new instance of the pipeline with the same component graph but with a different set of parameters. Not to be confused with python's __new__ method. |
| *parameters* | Parameter dictionary for this pipeline. |
| *predict* | Make predictions using selected features. |
| *save* | Saves pipeline at file path. |
| *score* | Evaluate model performance on current and additional objectives. |
| *summary* | A short summary of the pipeline structure, describing the list of components used. |
| *transform* | Transform the input. |
| *transform_all_but_final* | Transforms the data by applying all pre-processing components. |

**can_tune_threshold_with_objective**(*self*, *objective*)

Determine whether the threshold of a binary classification pipeline can be tuned.

**Parameters** **objective** (*ObjectiveBase*) – Primary AutoMLSearch objective.

**Returns** True if the pipeline threshold can be tuned.

**Return type** bool

**clone**(*self*)

Constructs a new pipeline with the same components, parameters, and random seed.

**Returns** A new instance of this pipeline with identical components, parameters, and random seed.

**static create_objectives**(*objectives*)

Create objective instances from a list of strings or objective classes.

**property custom_name**(*self*)

Custom name of the pipeline.

**describe**(*self*, *return_dict=False*)

Outputs pipeline details including component parameters.

**Parameters** **return_dict** (*bool*) – If True, return dictionary of information about pipeline. Defaults to False.

**Returns** Dictionary of all component parameters if return_dict is True, else None.

**Return type** dict

**property feature_importance**(*self*)

Importance associated with each feature. Features dropped by the feature selection are excluded.

**Returns** Feature names and their corresponding importance

**Return type** pd.DataFrame

**fit**(*self*, *X*, *y*)

Build a regression model.

**Parameters**

- **X** (*pd.DataFrame or np.ndarray*) – The input training data of shape [n_samples, n_features]
- **y** (*pd.Series, np.ndarray*) – The target training data of length [n_samples]

**Returns** self

**Raises** **ValueError** – If the target is not numeric.

**fit_transform**(*self*, *X*, *y*)

Fit and transform all components in the component graph, if all components are Transformers.

**Parameters**

- **X** (*pd.DataFrame*) – Input features of shape [n_samples, n_features].
- **y** (*pd.Series*) – The target data of length [n_samples].

**Returns** Transformed output.

**Return type** pd.DataFrame

**Raises** **ValueError** – If final component is an Estimator.

**get_component**(*self*, *name*)

Returns component by name.

**Parameters** **name** (*str*) – Name of component.

**Returns** Component to return

**Return type** Component

**get_hyperparameter_ranges**(*self*, *custom_hyperparameters*)

Returns hyperparameter ranges from all components as a dictionary.

**Parameters** **custom_hyperparameters** (*dict*) – Custom hyperparameters for the pipeline.

**Returns** Dictionary of hyperparameter ranges for each component in the pipeline.

**Return type** dict

**graph**(*self*, *filepath=None*)

Generate an image representing the pipeline graph.

**Parameters** **filepath** (*str, optional*) – Path to where the graph should be saved. If set to None (as by default), the graph will not be saved.

**Returns** Graph object that can be directly displayed in Jupyter notebooks.

**Return type** graphviz.Digraph

**Raises**

- **RuntimeError** – If graphviz is not installed.
- **ValueError** – If path is not writeable.

**graph_dict**(*self*)

Generates a dictionary with nodes consisting of the component names and parameters, and edges detailing component relationships. This dictionary is JSON serializable in most cases.

x_edges specifies from which component feature data is being passed. y_edges specifies from which component target data is being passed. This can be used to build graphs across a variety of visualization tools. Template: {"Nodes": {"component_name": {"Name": class_name, "Parameters": parameters_attributes}, ...}}, "x_edges": [[from_component_name, to_component_name], [from_component_name, to_component_name], ...], "y_edges": [[from_component_name, to_component_name], [from_component_name, to_component_name], ...]}

**Returns** A dictionary representing the DAG structure.

**Return type** dag_dict (dict)

**graph_feature_importance**(*self*, *importance_threshold=0*)

Generate a bar graph of the pipeline's feature importance.

**Parameters** **importance_threshold** (*float, optional*) – If provided, graph features with a permutation importance whose absolute value is larger than importance_threshold. Defaults to zero.

**Returns** A bar graph showing features and their corresponding importance.

**Return type** plotly.Figure

**Raises** **ValueError** – If importance threshold is not valid.

**inverse_transform**(*self*, *y*)

Apply component inverse_transform methods to estimator predictions in reverse order.

Components that implement inverse_transform are PolynomialDecomposer, LogTransformer, LabelEncoder (tbd).

**Parameters** **y** (*pd.Series*) – Final component features.

**Returns** The inverse transform of the target.

**Return type** pd.Series

**static load**(*file_path*)

Loads pipeline at file path.

**Parameters** **file_path** (*str*) – Location to load file.

**Returns** PipelineBase object

**property model_family**(*self*)

Returns model family of this pipeline.

**property name**(*self*)

Name of the pipeline.

**new**(*self*, *parameters*, *random_seed=0*)

Constructs a new instance of the pipeline with the same component graph but with a different set of parameters. Not to be confused with python's __new__ method.

**Parameters**

- **parameters** (*dict*) – Dictionary with component names as keys and dictionary of that component's parameters as values. An empty dictionary or None implies using all default values for component parameters. Defaults to None.
- **random_seed** (*int*) – Seed for the random number generator. Defaults to 0.

**Returns** A new instance of this pipeline with identical components.

**property parameters**(*self*)

Parameter dictionary for this pipeline.

**Returns** Dictionary of all component parameters.

**Return type** dict

**predict**(*self*, *X*, *objective=None*, *X_train=None*, *y_train=None*)

Make predictions using selected features.

**Parameters**

- **X** (*pd.DataFrame, or np.ndarray*) – Data of shape [n_samples, n_features].
- **objective** (*Object or string*) – The objective to use to make predictions.
- **X_train** (*pd.DataFrame or np.ndarray or None*) – Training data. Ignored. Only used for time series.
- **y_train** (*pd.Series or None*) – Training labels. Ignored. Only used for time series.

**Returns** Predicted values.

**Return type** pd.Series

**save**(*self*, *file_path*, *pickle_protocol=cloudpickle.DEFAULT_PROTOCOL*)

Saves pipeline at file path.

**Parameters**

- **file_path** (*str*) – Location to save file.
- **pickle_protocol** (*int*) – The pickle data stream format.

**score**(*self*, *X*, *y*, *objectives*, *X_train=None*, *y_train=None*)

Evaluate model performance on current and additional objectives.

**Parameters**

- **X** (*pd.DataFrame, or np.ndarray*) – Data of shape [n_samples, n_features]
- **y** (*pd.Series, or np.ndarray*) – True values of length [n_samples]
- **objectives** (*list*) – Non-empty list of objectives to score on
- **X_train** (*pd.DataFrame or np.ndarray or None*) – Training data. Ignored. Only used for time series.
- **y_train** (*pd.Series or None*) – Training labels. Ignored. Only used for time series.

**Returns** Ordered dictionary of objective scores.

**Return type** dict

**property summary**(*self*)

A short summary of the pipeline structure, describing the list of components used.

Example: Logistic Regression Classifier w/ Simple Imputer + One Hot Encoder

**Returns** A string describing the pipeline structure.

**transform**(*self*, *X*, *y=None*)

Transform the input.

**Parameters**

- **X** (*pd.DataFrame, or np.ndarray*) – Data of shape [n_samples, n_features].
- **y** (*pd.Series*) – The target data of length [n_samples]. Defaults to None.

**Returns** Transformed output.

**Return type** pd.DataFrame

**transform_all_but_final**(*self*, *X*, *y=None*, *X_train=None*, *y_train=None*)

Transforms the data by applying all pre-processing components.

**Parameters**

- **X** (*pd.DataFrame*) – Input data to the pipeline to transform.
- **y** (*pd.Series or None*) – Targets corresponding to X. Optional.
- **X_train** (*pd.DataFrame or np.ndarray or None*) – Training data. Only used for time series.
- **y_train** (*pd.Series or None*) – Training labels. Only used for time series.

**Returns** New transformed features.

**Return type** pd.DataFrame

### time_series_classification_pipelines

Pipeline base class for time-series classification problems.

### Module Contents

### Classes Summary

| | |
|---|---|
| *TimeSeriesBinaryClassificationPipeline* | Pipeline base class for time series binary classification problems. |
| *TimeSeriesClassificationPipeline* | Pipeline base class for time series classification problems. |
| *TimeSeriesMulticlassClassificationPipeline* | Pipeline base class for time series multiclass classification problems. |

### Contents

**class** evalml.pipelines.time_series_classification_pipelines.**TimeSeriesBinaryClassificationPipeline**(*comp*
*pa-*
*ram-*
*e-*
*ters=*
*cus-*
*tom_r*
*ran-*
*dom_*

Pipeline base class for time series binary classification problems.

**Parameters**

- **component_graph** (*list or dict*) – List of components in order. Accepts strings or ComponentBase subclasses in the list. Note that when duplicate components are specified in a list, the duplicate component names will be modified with the component's index in the list. For example, the component graph [Imputer, One Hot Encoder, Imputer, Logistic Regression Classifier] will have names ["Imputer", "One Hot Encoder", "Imputer_2", "Logistic Regression Classifier"]
- **parameters** (*dict*) – Dictionary with component names as keys and dictionary of that component's parameters as values. An empty dictionary {} implies using all default values for component parameters. Pipeline-level parameters such as time_index, gap, and max_delay must be specified with the "pipeline" key. For example: Pipeline(parameters={"pipeline": {"time_index": "Date", "max_delay": 4, "gap": 2}}).
- **random_seed** (*int*) – Seed for the random number generator. Defaults to 0.

### Example

```
>>> pipeline = TimeSeriesBinaryClassificationPipeline(component_graph=["Simple␣
↪Imputer", "Logistic Regression Classifier"],
...                                                parameters={"Logistic␣
↪Regression Classifier": {"penalty": "elasticnet",
...                                                                          ␣
↪                 "solver": "liblinear"},
...                                                                 "pipeline": {"gap
↪": 1, "max_delay": 1, "forecast_horizon": 1, "time_index": "date"}},
```

(continues on next page)

(continued from previous page)

```
...                                                custom_name="My␣
↪TimeSeriesBinary Pipeline")
...
>>> assert pipeline.custom_name == "My TimeSeriesBinary Pipeline"
>>> assert pipeline.component_graph.component_dict.keys() == {'Simple Imputer',
↪'Logistic Regression Classifier'}
...
>>> assert pipeline.parameters == {
...     'Simple Imputer': {'impute_strategy': 'most_frequent', 'fill_value': None},
...     'Logistic Regression Classifier': {'penalty': 'elasticnet',
...                                        'C': 1.0,
...                                        'n_jobs': -1,
...                                        'multi_class': 'auto',
...                                        'solver': 'liblinear'},
...     'pipeline': {'gap': 1, 'max_delay': 1, 'forecast_horizon': 1, 'time_index':
↪"date"}}
```

**Attributes**

| | |
|---|---|
| **problem_type** | None |

**Methods**

| | |
|---|---|
| *can_tune_threshold_with_objective* | Determine whether the threshold of a binary classification pipeline can be tuned. |
| *classes_* | Gets the class names for the pipeline. Will return None before pipeline is fit. |
| *clone* | Constructs a new pipeline with the same components, parameters, and random seed. |
| *create_objectives* | Create objective instances from a list of strings or objective classes. |
| *custom_name* | Custom name of the pipeline. |
| *dates_needed_for_prediction* | Return dates needed to forecast the given date in the future. |
| *dates_needed_for_prediction_range* | Return dates needed to forecast the given date in the future. |
| *describe* | Outputs pipeline details including component parameters. |
| *feature_importance* | Importance associated with each feature. Features dropped by the feature selection are excluded. |
| *fit* | Fit a time series classification model. |
| *fit_transform* | Fit and transform all components in the component graph, if all components are Transformers. |
| *get_component* | Returns component by name. |
| *get_hyperparameter_ranges* | Returns hyperparameter ranges from all components as a dictionary. |
| *graph* | Generate an image representing the pipeline graph. |

continues on next page

Table 7 – continued from previous page

| | |
|---|---|
| *graph_dict* | Generates a dictionary with nodes consisting of the component names and parameters, and edges detailing component relationships. This dictionary is JSON serializable in most cases. |
| *graph_feature_importance* | Generate a bar graph of the pipeline's feature importance. |
| *inverse_transform* | Apply component inverse_transform methods to estimator predictions in reverse order. |
| *load* | Loads pipeline at file path. |
| *model_family* | Returns model family of this pipeline. |
| *name* | Name of the pipeline. |
| *new* | Constructs a new instance of the pipeline with the same component graph but with a different set of parameters. Not to be confused with python's __new__ method. |
| *optimize_threshold* | Optimize the pipeline threshold given the objective to use. Only used for binary problems with objectives whose thresholds can be tuned. |
| *parameters* | Parameter dictionary for this pipeline. |
| *predict* | Predict on future data where target is not known. |
| *predict_in_sample* | Predict on future data where the target is known, e.g. cross validation. |
| *predict_proba* | Predict on future data where the target is unknown. |
| *predict_proba_in_sample* | Predict on future data where the target is known, e.g. cross validation. |
| *save* | Saves pipeline at file path. |
| *score* | Evaluate model performance on current and additional objectives. |
| *summary* | A short summary of the pipeline structure, describing the list of components used. |
| *threshold* | Threshold used to make a prediction. Defaults to None. |
| *transform* | Transform the input. |
| *transform_all_but_final* | Transforms the data by applying all pre-processing components. |

**can_tune_threshold_with_objective**(*self*, *objective*)

Determine whether the threshold of a binary classification pipeline can be tuned.

**Parameters** **objective** (*ObjectiveBase*) – Primary AutoMLSearch objective.

**Returns** True if the pipeline threshold can be tuned.

**Return type** bool

**property classes_**(*self*)

Gets the class names for the pipeline. Will return None before pipeline is fit.

**clone**(*self*)

Constructs a new pipeline with the same components, parameters, and random seed.

**Returns** A new instance of this pipeline with identical components, parameters, and random seed.

**static create_objectives**(*objectives*)

Create objective instances from a list of strings or objective classes.

**property custom_name**(*self*)

Custom name of the pipeline.

**dates_needed_for_prediction**(*self*, *date*)

Return dates needed to forecast the given date in the future.

**Parameters** **date** (*pd.Timestamp*) – Date to forecast in the future.

**Returns** Range of dates needed to forecast the given date.

**Return type** dates_needed (tuple(pd.Timestamp))

**dates_needed_for_prediction_range**(*self*, *start_date*, *end_date*)

Return dates needed to forecast the given date in the future.

**Parameters**

- **start_date** (*pd.Timestamp*) – Start date of range to forecast in the future.
- **end_date** (*pd.Timestamp*) – End date of range to forecast in the future.

**Returns** Range of dates needed to forecast the given date.

**Return type** dates_needed (tuple(pd.Timestamp))

**Raises** **ValueError** – If start_date doesn't come before end_date

**describe**(*self*, *return_dict=False*)

Outputs pipeline details including component parameters.

**Parameters** **return_dict** (*bool*) – If True, return dictionary of information about pipeline. Defaults to False.

**Returns** Dictionary of all component parameters if return_dict is True, else None.

**Return type** dict

**property feature_importance**(*self*)

Importance associated with each feature. Features dropped by the feature selection are excluded.

**Returns** Feature names and their corresponding importance

**Return type** pd.DataFrame

**fit**(*self*, *X*, *y*)

Fit a time series classification model.

**Parameters**

- **X** (*pd.DataFrame or np.ndarray*) – The input training data of shape [n_samples, n_features]
- **y** (*pd.Series, np.ndarray*) – The target training labels of length [n_samples]

**Returns** self

**Raises** **ValueError** – If the number of unique classes in y are not appropriate for the type of pipeline.

**fit_transform**(*self*, *X*, *y*)

Fit and transform all components in the component graph, if all components are Transformers.

**Parameters**

- **X** (*pd.DataFrame*) – Input features of shape [n_samples, n_features].
- **y** (*pd.Series*) – The target data of length [n_samples].

**Returns** Transformed output.

**Return type** pd.DataFrame

**Raises** **ValueError** – If final component is an Estimator.

**get_component**(*self*, *name*)

Returns component by name.

**Parameters** **name** (*str*) – Name of component.

**Returns** Component to return

**Return type** Component

**get_hyperparameter_ranges**(*self*, *custom_hyperparameters*)

Returns hyperparameter ranges from all components as a dictionary.

**Parameters** **custom_hyperparameters** (*dict*) – Custom hyperparameters for the pipeline.

**Returns** Dictionary of hyperparameter ranges for each component in the pipeline.

**Return type** dict

**graph**(*self*, *filepath=None*)

Generate an image representing the pipeline graph.

**Parameters** **filepath** (*str, optional*) – Path to where the graph should be saved. If set to None (as by default), the graph will not be saved.

**Returns** Graph object that can be directly displayed in Jupyter notebooks.

**Return type** graphviz.Digraph

**Raises**

- **RuntimeError** – If graphviz is not installed.
- **ValueError** – If path is not writeable.

**graph_dict**(*self*)

Generates a dictionary with nodes consisting of the component names and parameters, and edges detailing component relationships. This dictionary is JSON serializable in most cases.

x_edges specifies from which component feature data is being passed. y_edges specifies from which component target data is being passed. This can be used to build graphs across a variety of visualization tools. Template: {"Nodes": {"component_name": {"Name": class_name, "Parameters": parameters_attributes}, ...}}, "x_edges": [[from_component_name, to_component_name], [from_component_name, to_component_name], ...], "y_edges": [[from_component_name, to_component_name], [from_component_name, to_component_name], ...]}

**Returns** A dictionary representing the DAG structure.

**Return type** dag_dict (dict)

**graph_feature_importance**(*self*, *importance_threshold=0*)

Generate a bar graph of the pipeline's feature importance.

**Parameters** **importance_threshold** (*float, optional*) – If provided, graph features with a permutation importance whose absolute value is larger than importance_threshold. Defaults to zero.

**Returns** A bar graph showing features and their corresponding importance.

**Return type** plotly.Figure

**Raises** **ValueError** – If importance threshold is not valid.

**inverse_transform**(*self*, *y*)

Apply component inverse_transform methods to estimator predictions in reverse order.

Components that implement inverse_transform are PolynomialDecomposer, LogTransformer, LabelEncoder (tbd).

**Parameters** **y** (*pd.Series*) – Final component features.

**Returns** The inverse transform of the target.

**Return type** pd.Series

**static load**(*file_path*)

Loads pipeline at file path.

**Parameters** **file_path** (*str*) – Location to load file.

**Returns** PipelineBase object

**property model_family**(*self*)

Returns model family of this pipeline.

**property name**(*self*)

Name of the pipeline.

**new**(*self*, *parameters*, *random_seed=0*)

Constructs a new instance of the pipeline with the same component graph but with a different set of parameters. Not to be confused with python's __new__ method.

**Parameters**

- **parameters** (*dict*) – Dictionary with component names as keys and dictionary of that component's parameters as values. An empty dictionary or None implies using all default values for component parameters. Defaults to None.
- **random_seed** (*int*) – Seed for the random number generator. Defaults to 0.

**Returns** A new instance of this pipeline with identical components.

**optimize_threshold**(*self*, *X*, *y*, *y_pred_proba*, *objective*)

Optimize the pipeline threshold given the objective to use. Only used for binary problems with objectives whose thresholds can be tuned.

**Parameters**

- **X** (*pd.DataFrame*) – Input features.
- **y** (*pd.Series*) – Input target values.
- **y_pred_proba** (*pd.Series*) – The predicted probabilities of the target outputted by the pipeline.

- **objective** (*ObjectiveBase*) – The objective to threshold with. Must have a tunable threshold.

**Raises** **ValueError** – If objective is not optimizable.

**property parameters**(*self*)

Parameter dictionary for this pipeline.

**Returns** Dictionary of all component parameters.

**Return type** dict

**predict**(*self*, *X*, *objective=None*, *X_train=None*, *y_train=None*)

Predict on future data where target is not known.

**Parameters**

- **X** (*pd.DataFrame, or np.ndarray*) – Data of shape [n_samples, n_features].
- **objective** (*Object or string*) – The objective to use to make predictions.
- **X_train** (*pd.DataFrame or np.ndarray or None*) – Training data.
- **y_train** (*pd.Series or None*) – Training labels.

**Raises** **ValueError** – If X_train and/or y_train are None or if final component is not an Estimator.

**Returns** Predictions.

**predict_in_sample**(*self*, *X*, *y*, *X_train*, *y_train*, *objective=None*)

Predict on future data where the target is known, e.g. cross validation.

**Parameters**

- **X** (*pd.DataFrame*) – Future data of shape [n_samples, n_features].
- **y** (*pd.Series*) – Future target of shape [n_samples].
- **X_train** (*pd.DataFrame*) – Data the pipeline was trained on of shape [n_samples_train, n_feautures].
- **y_train** (*pd.Series*) – Targets used to train the pipeline of shape [n_samples_train].
- **objective** (*ObjectiveBase, str*) – Objective used to threshold predicted probabilities, optional. Defaults to None.

**Returns** Estimated labels.

**Return type** pd.Series

**Raises** **ValueError** – If objective is not defined for time-series binary classification problems.

**predict_proba**(*self*, *X*, *X_train=None*, *y_train=None*)

Predict on future data where the target is unknown.

**Parameters**

- **X** (*pd.DataFrame or np.ndarray*) – Future data of shape [n_samples, n_features].
- **X_train** (*pd.DataFrame, np.ndarray*) – Data the pipeline was trained on of shape [n_samples_train, n_features].
- **y_train** (*pd.Series, np.ndarray*) – Targets used to train the pipeline of shape [n_samples_train].

**Returns** Estimated probabilities.

**Return type** pd.Series

**Raises** `ValueError` – If final component is not an Estimator.

`predict_proba_in_sample`(*self*, *X_holdout*, *y_holdout*, *X_train*, *y_train*)

Predict on future data where the target is known, e.g. cross validation.

**Parameters**

- `X_holdout` (*pd.DataFrame or np.ndarray*) – Future data of shape [n_samples, n_features].
- `y_holdout` (*pd.Series, np.ndarray*) – Future target of shape [n_samples].
- `X_train` (*pd.DataFrame, np.ndarray*) – Data the pipeline was trained on of shape [n_samples_train, n_features].
- `y_train` (*pd.Series, np.ndarray*) – Targets used to train the pipeline of shape [n_samples_train].

**Returns** Estimated probabilities.

**Return type** pd.Series

**Raises** `ValueError` – If the final component is not an Estimator.

`save`(*self*, *file_path*, *pickle_protocol=cloudpickle.DEFAULT_PROTOCOL*)

Saves pipeline at file path.

**Parameters**

- `file_path` (*str*) – Location to save file.
- `pickle_protocol` (*int*) – The pickle data stream format.

`score`(*self*, *X*, *y*, *objectives*, *X_train=None*, *y_train=None*)

Evaluate model performance on current and additional objectives.

**Parameters**

- `X` (*pd.DataFrame or np.ndarray*) – Data of shape [n_samples, n_features].
- `y` (*pd.Series*) – True labels of length [n_samples].
- `objectives` (*list*) – Non-empty list of objectives to score on.
- `X_train` (*pd.DataFrame, np.ndarray*) – Data the pipeline was trained on of shape [n_samples_train, n_features].
- `y_train` (*pd.Series, np.ndarray*) – Targets used to train the pipeline of shape [n_samples_train].

**Returns** Ordered dictionary of objective scores.

**Return type** dict

`property summary`(*self*)

A short summary of the pipeline structure, describing the list of components used.

Example: Logistic Regression Classifier w/ Simple Imputer + One Hot Encoder

**Returns** A string describing the pipeline structure.

`property threshold`(*self*)

Threshold used to make a prediction. Defaults to None.

**transform**(*self*, *X*, *y=None*)

Transform the input.

**Parameters**

- **X** (*pd.DataFrame, or np.ndarray*) – Data of shape [n_samples, n_features].
- **y** (*pd.Series*) – The target data of length [n_samples]. Defaults to None.

**Returns** Transformed output.

**Return type** pd.DataFrame

**transform_all_but_final**(*self*, *X*, *y=None*, *X_train=None*, *y_train=None*, *calculating_residuals=False*)

Transforms the data by applying all pre-processing components.

**Parameters**

- **X** (*pd.DataFrame*) – Input data to the pipeline to transform.
- **y** (*pd.Series*) – Targets corresponding to the pipeline targets.
- **X_train** (*pd.DataFrame*) – Training data used to generate generates from past observations.
- **y_train** (*pd.Series*) – Training targets used to generate features from past observations.
- **calculating_residuals** (*bool*) – Whether we're calling predict_in_sample to calculate the residuals. This means the X and y arguments are not future data, but actually the train data.

**Returns** New transformed features.

**Return type** pd.DataFrame

**class** evalml.pipelines.time_series_classification_pipelines.**TimeSeriesClassificationPipeline**(*component_g*
*pa-*
*ram-*
*e-*
*ters=None*,
*cus-*
*tom_name=N*
*ran-*
*dom_seed=0*

Pipeline base class for time series classification problems.

**Parameters**

- **component_graph** (*ComponentGraph, list, dict*) – ComponentGraph instance, list of components in order, or dictionary of components. Accepts strings or ComponentBase subclasses in the list. Note that when duplicate components are specified in a list, the duplicate component names will be modified with the component's index in the list. For example, the component graph [Imputer, One Hot Encoder, Imputer, Logistic Regression Classifier] will have names ["Imputer", "One Hot Encoder", "Imputer_2", "Logistic Regression Classifier"]
- **parameters** (*dict*) – Dictionary with component names as keys and dictionary of that component's parameters as values. An empty dictionary {} implies using all default values for component parameters. Pipeline-level parameters such as time_index, gap, and max_delay must be specified with the "pipeline" key. For example: Pipeline(parameters={"pipeline": {"time_index": "Date", "max_delay": 4, "gap": 2}}).
- **random_seed** (*int*) – Seed for the random number generator. Defaults to 0.

**Attributes**

| | |
|---|---|
| **problem_type** | None |

**Methods**

| | |
|---|---|
| *can_tune_threshold_with_objective* | Determine whether the threshold of a binary classification pipeline can be tuned. |
| *classes_* | Gets the class names for the pipeline. Will return None before pipeline is fit. |
| *clone* | Constructs a new pipeline with the same components, parameters, and random seed. |
| *create_objectives* | Create objective instances from a list of strings or objective classes. |
| *custom_name* | Custom name of the pipeline. |
| *dates_needed_for_prediction* | Return dates needed to forecast the given date in the future. |
| *dates_needed_for_prediction_range* | Return dates needed to forecast the given date in the future. |
| *describe* | Outputs pipeline details including component parameters. |
| *feature_importance* | Importance associated with each feature. Features dropped by the feature selection are excluded. |
| *fit* | Fit a time series classification model. |
| *fit_transform* | Fit and transform all components in the component graph, if all components are Transformers. |
| *get_component* | Returns component by name. |
| *get_hyperparameter_ranges* | Returns hyperparameter ranges from all components as a dictionary. |
| *graph* | Generate an image representing the pipeline graph. |
| *graph_dict* | Generates a dictionary with nodes consisting of the component names and parameters, and edges detailing component relationships. This dictionary is JSON serializable in most cases. |
| *graph_feature_importance* | Generate a bar graph of the pipeline's feature importance. |
| *inverse_transform* | Apply component inverse_transform methods to estimator predictions in reverse order. |
| *load* | Loads pipeline at file path. |
| *model_family* | Returns model family of this pipeline. |
| *name* | Name of the pipeline. |
| *new* | Constructs a new instance of the pipeline with the same component graph but with a different set of parameters. Not to be confused with python's __new__ method. |
| *parameters* | Parameter dictionary for this pipeline. |
| *predict* | Predict on future data where target is not known. |
| *predict_in_sample* | Predict on future data where the target is known, e.g. cross validation. |
| *predict_proba* | Predict on future data where the target is unknown. |

continues on next page

Table 8 – continued from previous page

| | |
|---|---|
| *predict_proba_in_sample* | Predict on future data where the target is known, e.g. cross validation. |
| *save* | Saves pipeline at file path. |
| *score* | Evaluate model performance on current and additional objectives. |
| *summary* | A short summary of the pipeline structure, describing the list of components used. |
| *transform* | Transform the input. |
| *transform_all_but_final* | Transforms the data by applying all pre-processing components. |

**can_tune_threshold_with_objective**(*self*, *objective*)

Determine whether the threshold of a binary classification pipeline can be tuned.

**Parameters** **objective** (*ObjectiveBase*) – Primary AutoMLSearch objective.

**Returns** True if the pipeline threshold can be tuned.

**Return type** bool

**property classes_**(*self*)

Gets the class names for the pipeline. Will return None before pipeline is fit.

**clone**(*self*)

Constructs a new pipeline with the same components, parameters, and random seed.

**Returns** A new instance of this pipeline with identical components, parameters, and random seed.

**static create_objectives**(*objectives*)

Create objective instances from a list of strings or objective classes.

**property custom_name**(*self*)

Custom name of the pipeline.

**dates_needed_for_prediction**(*self*, *date*)

Return dates needed to forecast the given date in the future.

**Parameters** **date** (*pd.Timestamp*) – Date to forecast in the future.

**Returns** Range of dates needed to forecast the given date.

**Return type** dates_needed (tuple(pd.Timestamp))

**dates_needed_for_prediction_range**(*self*, *start_date*, *end_date*)

Return dates needed to forecast the given date in the future.

**Parameters**

- **start_date** (*pd.Timestamp*) – Start date of range to forecast in the future.
- **end_date** (*pd.Timestamp*) – End date of range to forecast in the future.

**Returns** Range of dates needed to forecast the given date.

**Return type** dates_needed (tuple(pd.Timestamp))

**Raises** **ValueError** – If start_date doesn't come before end_date

**describe**(*self*, *return_dict=False*)

Outputs pipeline details including component parameters.

**Parameters** **return_dict** (*bool*) – If True, return dictionary of information about pipeline. Defaults to False.

**Returns** Dictionary of all component parameters if return_dict is True, else None.

**Return type** dict

**property feature_importance**(*self*)

Importance associated with each feature. Features dropped by the feature selection are excluded.

**Returns** Feature names and their corresponding importance

**Return type** pd.DataFrame

**fit**(*self*, *X*, *y*)

Fit a time series classification model.

**Parameters**

- **X** (*pd.DataFrame or np.ndarray*) – The input training data of shape [n_samples, n_features]
- **y** (*pd.Series, np.ndarray*) – The target training labels of length [n_samples]

**Returns** self

**Raises** **ValueError** – If the number of unique classes in y are not appropriate for the type of pipeline.

**fit_transform**(*self*, *X*, *y*)

Fit and transform all components in the component graph, if all components are Transformers.

**Parameters**

- **X** (*pd.DataFrame*) – Input features of shape [n_samples, n_features].
- **y** (*pd.Series*) – The target data of length [n_samples].

**Returns** Transformed output.

**Return type** pd.DataFrame

**Raises** **ValueError** – If final component is an Estimator.

**get_component**(*self*, *name*)

Returns component by name.

**Parameters** **name** (*str*) – Name of component.

**Returns** Component to return

**Return type** Component

**get_hyperparameter_ranges**(*self*, *custom_hyperparameters*)

Returns hyperparameter ranges from all components as a dictionary.

**Parameters** **custom_hyperparameters** (*dict*) – Custom hyperparameters for the pipeline.

**Returns** Dictionary of hyperparameter ranges for each component in the pipeline.

**Return type** dict

**graph**(*self*, *filepath=None*)

Generate an image representing the pipeline graph.

**Parameters** **filepath** (*str, optional*) – Path to where the graph should be saved. If set to None (as by default), the graph will not be saved.

**Returns** Graph object that can be directly displayed in Jupyter notebooks.

**Return type** graphviz.Digraph

**Raises**

- **RuntimeError** – If graphviz is not installed.
- **ValueError** – If path is not writeable.

**graph_dict**(*self*)

Generates a dictionary with nodes consisting of the component names and parameters, and edges detailing component relationships. This dictionary is JSON serializable in most cases.

x_edges specifies from which component feature data is being passed. y_edges specifies from which component target data is being passed. This can be used to build graphs across a variety of visualization tools. Template: {"Nodes": {"component_name": {"Name": class_name, "Parameters": parameters_attributes}, ...}}, "x_edges": [[from_component_name, to_component_name], [from_component_name, to_component_name], ...], "y_edges": [[from_component_name, to_component_name], [from_component_name, to_component_name], ...]}

**Returns** A dictionary representing the DAG structure.

**Return type** dag_dict (dict)

**graph_feature_importance**(*self*, *importance_threshold=0*)

Generate a bar graph of the pipeline's feature importance.

**Parameters** **importance_threshold** (*float, optional*) – If provided, graph features with a permutation importance whose absolute value is larger than importance_threshold. Defaults to zero.

**Returns** A bar graph showing features and their corresponding importance.

**Return type** plotly.Figure

**Raises** **ValueError** – If importance threshold is not valid.

**inverse_transform**(*self*, *y*)

Apply component inverse_transform methods to estimator predictions in reverse order.

Components that implement inverse_transform are PolynomialDecomposer, LogTransformer, LabelEncoder (tbd).

**Parameters** **y** (*pd.Series*) – Final component features.

**Returns** The inverse transform of the target.

**Return type** pd.Series

**static load**(*file_path*)

Loads pipeline at file path.

**Parameters** **file_path** (*str*) – Location to load file.

**Returns** PipelineBase object

**property model_family**(*self*)

Returns model family of this pipeline.

**property name**(*self*)

Name of the pipeline.

**new**(*self*, *parameters*, *random_seed=0*)

Constructs a new instance of the pipeline with the same component graph but with a different set of parameters. Not to be confused with python's __new__ method.

**Parameters**

- **parameters** (*dict*) – Dictionary with component names as keys and dictionary of that component's parameters as values. An empty dictionary or None implies using all default values for component parameters. Defaults to None.
- **random_seed** (*int*) – Seed for the random number generator. Defaults to 0.

**Returns** A new instance of this pipeline with identical components.

**property parameters**(*self*)

Parameter dictionary for this pipeline.

**Returns** Dictionary of all component parameters.

**Return type** dict

**predict**(*self*, *X*, *objective=None*, *X_train=None*, *y_train=None*)

Predict on future data where target is not known.

**Parameters**

- **X** (*pd.DataFrame, or np.ndarray*) – Data of shape [n_samples, n_features].
- **objective** (*Object or string*) – The objective to use to make predictions.
- **X_train** (*pd.DataFrame or np.ndarray or None*) – Training data.
- **y_train** (*pd.Series or None*) – Training labels.

**Raises** **ValueError** – If X_train and/or y_train are None or if final component is not an Estimator.

**Returns** Predictions.

**predict_in_sample**(*self*, *X*, *y*, *X_train*, *y_train*, *objective=None*)

Predict on future data where the target is known, e.g. cross validation.

Note: we cast y as ints first to address boolean values that may be returned from calculating predictions which we would not be able to otherwise transform if we originally had integer targets.

**Parameters**

- **X** (*pd.DataFrame or np.ndarray*) – Future data of shape [n_samples, n_features].
- **y** (*pd.Series, np.ndarray*) – Future target of shape [n_samples].
- **X_train** (*pd.DataFrame, np.ndarray*) – Data the pipeline was trained on of shape [n_samples_train, n_features].
- **y_train** (*pd.Series, np.ndarray*) – Targets used to train the pipeline of shape [n_samples_train].
- **objective** (*ObjectiveBase, str, None*) – Objective used to threshold predicted probabilities, optional.

**Returns** Estimated labels.

**Return type** pd.Series

**Raises** `ValueError` – If final component is not an Estimator.

`predict_proba`(*self*, *X*, *X_train=None*, *y_train=None*)

Predict on future data where the target is unknown.

**Parameters**

- **X** (*pd.DataFrame or np.ndarray*) – Future data of shape [n_samples, n_features].
- **X_train** (*pd.DataFrame, np.ndarray*) – Data the pipeline was trained on of shape [n_samples_train, n_features].
- **y_train** (*pd.Series, np.ndarray*) – Targets used to train the pipeline of shape [n_samples_train].

**Returns** Estimated probabilities.

**Return type** pd.Series

**Raises** `ValueError` – If final component is not an Estimator.

`predict_proba_in_sample`(*self*, *X_holdout*, *y_holdout*, *X_train*, *y_train*)

Predict on future data where the target is known, e.g. cross validation.

**Parameters**

- **X_holdout** (*pd.DataFrame or np.ndarray*) – Future data of shape [n_samples, n_features].
- **y_holdout** (*pd.Series, np.ndarray*) – Future target of shape [n_samples].
- **X_train** (*pd.DataFrame, np.ndarray*) – Data the pipeline was trained on of shape [n_samples_train, n_features].
- **y_train** (*pd.Series, np.ndarray*) – Targets used to train the pipeline of shape [n_samples_train].

**Returns** Estimated probabilities.

**Return type** pd.Series

**Raises** `ValueError` – If the final component is not an Estimator.

`save`(*self*, *file_path*, *pickle_protocol=cloudpickle.DEFAULT_PROTOCOL*)

Saves pipeline at file path.

**Parameters**

- **file_path** (*str*) – Location to save file.
- **pickle_protocol** (*int*) – The pickle data stream format.

`score`(*self*, *X*, *y*, *objectives*, *X_train=None*, *y_train=None*)

Evaluate model performance on current and additional objectives.

**Parameters**

- **X** (*pd.DataFrame or np.ndarray*) – Data of shape [n_samples, n_features].
- **y** (*pd.Series*) – True labels of length [n_samples].
- **objectives** (*list*) – Non-empty list of objectives to score on.

- **X_train** (*pd.DataFrame, np.ndarray*) – Data the pipeline was trained on of shape [n_samples_train, n_features].
- **y_train** (*pd.Series, np.ndarray*) – Targets used to train the pipeline of shape [n_samples_train].

**Returns** Ordered dictionary of objective scores.

**Return type** dict

**property summary**(*self*)

A short summary of the pipeline structure, describing the list of components used.

Example: Logistic Regression Classifier w/ Simple Imputer + One Hot Encoder

**Returns** A string describing the pipeline structure.

**transform**(*self*, *X*, *y=None*)

Transform the input.

**Parameters**

- **X** (*pd.DataFrame, or np.ndarray*) – Data of shape [n_samples, n_features].
- **y** (*pd.Series*) – The target data of length [n_samples]. Defaults to None.

**Returns** Transformed output.

**Return type** pd.DataFrame

**transform_all_but_final**(*self*, *X*, *y=None*, *X_train=None*, *y_train=None*, *calculating_residuals=False*)

Transforms the data by applying all pre-processing components.

**Parameters**

- **X** (*pd.DataFrame*) – Input data to the pipeline to transform.
- **y** (*pd.Series*) – Targets corresponding to the pipeline targets.
- **X_train** (*pd.DataFrame*) – Training data used to generate generates from past observations.
- **y_train** (*pd.Series*) – Training targets used to generate features from past observations.
- **calculating_residuals** (*bool*) – Whether we're calling predict_in_sample to calculate the residuals. This means the X and y arguments are not future data, but actually the train data.

**Returns** New transformed features.

**Return type** pd.DataFrame

**class** evalml.pipelines.time_series_classification_pipelines.**TimeSeriesMulticlassClassificationPipeline**(

Pipeline base class for time series multiclass classification problems.

**Parameters**

- **component_graph** (*list or dict*) – List of components in order. Accepts strings or ComponentBase subclasses in the list. Note that when duplicate components are specified in a list, the duplicate component names will be modified with the component's index in the list. For example, the component graph [Imputer, One Hot Encoder, Imputer, Logistic Regression Classifier] will have names ["Imputer", "One Hot Encoder", "Imputer_2", "Logistic Regression Classifier"]
- **parameters** (*dict*) – Dictionary with component names as keys and dictionary of that component's parameters as values. An empty dictionary {} implies using all default values for component parameters. Pipeline-level parameters such as time_index, gap, and max_delay must be specified with the "pipeline" key. For example: Pipeline(parameters={"pipeline": {"time_index": "Date", "max_delay": 4, "gap": 2}}).
- **random_seed** (*int*) – Seed for the random number generator. Defaults to 0.

### Example

```
>>> pipeline = TimeSeriesMulticlassClassificationPipeline(component_graph=["Simple␣
↪Imputer", "Logistic Regression Classifier"],
...                                                    parameters={"Logistic␣
↪Regression Classifier": {"penalty": "elasticnet",
...                                                                                ␣
↪                     "solver": "liblinear"},
...                                                                  "pipeline": {
↪"gap": 1, "max_delay": 1, "forecast_horizon": 1, "time_index": "date"}},
...                                                    custom_name="My␣
↪TimeSeriesMulticlass Pipeline")
>>> assert pipeline.custom_name == "My TimeSeriesMulticlass Pipeline"
>>> assert pipeline.component_graph.component_dict.keys() == {'Simple Imputer',
↪'Logistic Regression Classifier'}
>>> assert pipeline.parameters == {
...  'Simple Imputer': {'impute_strategy': 'most_frequent', 'fill_value': None},
...  'Logistic Regression Classifier': {'penalty': 'elasticnet',
...                                     'C': 1.0,
...                                     'n_jobs': -1,
...                                     'multi_class': 'auto',
...                                     'solver': 'liblinear'},
...     'pipeline': {'gap': 1, 'max_delay': 1, 'forecast_horizon': 1, 'time_index':
↪"date"}}
```

#### Attributes

| | |
|---|---|
| **problem_type** | ProblemTypes.TIME_SERIES_MULTICLASS |

#### Methods

| | |
|---|---|
| *can_tune_threshold_with_objective* | Determine whether the threshold of a binary classification pipeline can be tuned. |
| *classes_* | Gets the class names for the pipeline. Will return None before pipeline is fit. |

continues on next page

Table 9 – continued from previous page

| | |
|---|---|
| *clone* | Constructs a new pipeline with the same components, parameters, and random seed. |
| *create_objectives* | Create objective instances from a list of strings or objective classes. |
| *custom_name* | Custom name of the pipeline. |
| *dates_needed_for_prediction* | Return dates needed to forecast the given date in the future. |
| *dates_needed_for_prediction_range* | Return dates needed to forecast the given date in the future. |
| *describe* | Outputs pipeline details including component parameters. |
| *feature_importance* | Importance associated with each feature. Features dropped by the feature selection are excluded. |
| *fit* | Fit a time series classification model. |
| *fit_transform* | Fit and transform all components in the component graph, if all components are Transformers. |
| *get_component* | Returns component by name. |
| *get_hyperparameter_ranges* | Returns hyperparameter ranges from all components as a dictionary. |
| *graph* | Generate an image representing the pipeline graph. |
| *graph_dict* | Generates a dictionary with nodes consisting of the component names and parameters, and edges detailing component relationships. This dictionary is JSON serializable in most cases. |
| *graph_feature_importance* | Generate a bar graph of the pipeline's feature importance. |
| *inverse_transform* | Apply component inverse_transform methods to estimator predictions in reverse order. |
| *load* | Loads pipeline at file path. |
| *model_family* | Returns model family of this pipeline. |
| *name* | Name of the pipeline. |
| *new* | Constructs a new instance of the pipeline with the same component graph but with a different set of parameters. Not to be confused with python's __new__ method. |
| *parameters* | Parameter dictionary for this pipeline. |
| *predict* | Predict on future data where target is not known. |
| *predict_in_sample* | Predict on future data where the target is known, e.g. cross validation. |
| *predict_proba* | Predict on future data where the target is unknown. |
| *predict_proba_in_sample* | Predict on future data where the target is known, e.g. cross validation. |
| *save* | Saves pipeline at file path. |
| *score* | Evaluate model performance on current and additional objectives. |
| *summary* | A short summary of the pipeline structure, describing the list of components used. |
| *transform* | Transform the input. |
| *transform_all_but_final* | Transforms the data by applying all pre-processing components. |

**can_tune_threshold_with_objective**(*self*, *objective*)

Determine whether the threshold of a binary classification pipeline can be tuned.

**Parameters** **objective** (*ObjectiveBase*) – Primary AutoMLSearch objective.

**Returns** True if the pipeline threshold can be tuned.

**Return type** bool

**property classes_**(*self*)

Gets the class names for the pipeline. Will return None before pipeline is fit.

**clone**(*self*)

Constructs a new pipeline with the same components, parameters, and random seed.

**Returns** A new instance of this pipeline with identical components, parameters, and random seed.

**static create_objectives**(*objectives*)

Create objective instances from a list of strings or objective classes.

**property custom_name**(*self*)

Custom name of the pipeline.

**dates_needed_for_prediction**(*self*, *date*)

Return dates needed to forecast the given date in the future.

**Parameters** **date** (*pd.Timestamp*) – Date to forecast in the future.

**Returns** Range of dates needed to forecast the given date.

**Return type** dates_needed (tuple(pd.Timestamp))

**dates_needed_for_prediction_range**(*self*, *start_date*, *end_date*)

Return dates needed to forecast the given date in the future.

**Parameters**

- **start_date** (*pd.Timestamp*) – Start date of range to forecast in the future.
- **end_date** (*pd.Timestamp*) – End date of range to forecast in the future.

**Returns** Range of dates needed to forecast the given date.

**Return type** dates_needed (tuple(pd.Timestamp))

**Raises** **ValueError** – If start_date doesn't come before end_date

**describe**(*self*, *return_dict=False*)

Outputs pipeline details including component parameters.

**Parameters** **return_dict** (*bool*) – If True, return dictionary of information about pipeline. Defaults to False.

**Returns** Dictionary of all component parameters if return_dict is True, else None.

**Return type** dict

**property feature_importance**(*self*)

Importance associated with each feature. Features dropped by the feature selection are excluded.

**Returns** Feature names and their corresponding importance

**Return type** pd.DataFrame

**fit**(*self*, *X*, *y*)

Fit a time series classification model.

**Parameters**

- **X** (*pd.DataFrame or np.ndarray*) – The input training data of shape [n_samples, n_features]
- **y** (*pd.Series, np.ndarray*) – The target training labels of length [n_samples]

**Returns** self

**Raises** **ValueError** – If the number of unique classes in y are not appropriate for the type of pipeline.

**fit_transform**(*self*, *X*, *y*)

Fit and transform all components in the component graph, if all components are Transformers.

**Parameters**

- **X** (*pd.DataFrame*) – Input features of shape [n_samples, n_features].
- **y** (*pd.Series*) – The target data of length [n_samples].

**Returns** Transformed output.

**Return type** pd.DataFrame

**Raises** **ValueError** – If final component is an Estimator.

**get_component**(*self*, *name*)

Returns component by name.

**Parameters** **name** (*str*) – Name of component.

**Returns** Component to return

**Return type** Component

**get_hyperparameter_ranges**(*self*, *custom_hyperparameters*)

Returns hyperparameter ranges from all components as a dictionary.

**Parameters** **custom_hyperparameters** (*dict*) – Custom hyperparameters for the pipeline.

**Returns** Dictionary of hyperparameter ranges for each component in the pipeline.

**Return type** dict

**graph**(*self*, *filepath=None*)

Generate an image representing the pipeline graph.

**Parameters** **filepath** (*str, optional*) – Path to where the graph should be saved. If set to None (as by default), the graph will not be saved.

**Returns** Graph object that can be directly displayed in Jupyter notebooks.

**Return type** graphviz.Digraph

**Raises**

- **RuntimeError** – If graphviz is not installed.
- **ValueError** – If path is not writeable.

**graph_dict**(*self*)

Generates a dictionary with nodes consisting of the component names and parameters, and edges detailing component relationships. This dictionary is JSON serializable in most cases.

x_edges specifies from which component feature data is being passed. y_edges specifies from which component target data is being passed. This can be used to build graphs across a variety of visualization tools. Template: {"Nodes": {"component_name": {"Name": class_name, "Parameters": parameters_attributes}, ...}}, "x_edges": [[from_component_name, to_component_name], [from_component_name, to_component_name], ...], "y_edges": [[from_component_name, to_component_name], [from_component_name, to_component_name], ...]}

**Returns** A dictionary representing the DAG structure.

**Return type** dag_dict (dict)

**graph_feature_importance**(*self*, *importance_threshold=0*)

Generate a bar graph of the pipeline's feature importance.

**Parameters** **importance_threshold** (*float, optional*) – If provided, graph features with a permutation importance whose absolute value is larger than importance_threshold. Defaults to zero.

**Returns** A bar graph showing features and their corresponding importance.

**Return type** plotly.Figure

**Raises** **ValueError** – If importance threshold is not valid.

**inverse_transform**(*self*, *y*)

Apply component inverse_transform methods to estimator predictions in reverse order.

Components that implement inverse_transform are PolynomialDecomposer, LogTransformer, LabelEncoder (tbd).

**Parameters** **y** (*pd.Series*) – Final component features.

**Returns** The inverse transform of the target.

**Return type** pd.Series

**static load**(*file_path*)

Loads pipeline at file path.

**Parameters** **file_path** (*str*) – Location to load file.

**Returns** PipelineBase object

**property model_family**(*self*)

Returns model family of this pipeline.

**property name**(*self*)

Name of the pipeline.

**new**(*self*, *parameters*, *random_seed=0*)

Constructs a new instance of the pipeline with the same component graph but with a different set of parameters. Not to be confused with python's __new__ method.

**Parameters**

- **parameters** (*dict*) – Dictionary with component names as keys and dictionary of that component's parameters as values. An empty dictionary or None implies using all default values for component parameters. Defaults to None.

- **random_seed** (*int*) – Seed for the random number generator. Defaults to 0.

**Returns** A new instance of this pipeline with identical components.

**property parameters**(*self*)

Parameter dictionary for this pipeline.

**Returns** Dictionary of all component parameters.

**Return type** dict

**predict**(*self*, *X*, *objective=None*, *X_train=None*, *y_train=None*)

Predict on future data where target is not known.

**Parameters**

- **X** (*pd.DataFrame, or np.ndarray*) – Data of shape [n_samples, n_features].
- **objective** (*Object or string*) – The objective to use to make predictions.
- **X_train** (*pd.DataFrame or np.ndarray or None*) – Training data.
- **y_train** (*pd.Series or None*) – Training labels.

**Raises** **ValueError** – If X_train and/or y_train are None or if final component is not an Estimator.

**Returns** Predictions.

**predict_in_sample**(*self*, *X*, *y*, *X_train*, *y_train*, *objective=None*)

Predict on future data where the target is known, e.g. cross validation.

Note: we cast y as ints first to address boolean values that may be returned from calculating predictions which we would not be able to otherwise transform if we originally had integer targets.

**Parameters**

- **X** (*pd.DataFrame or np.ndarray*) – Future data of shape [n_samples, n_features].
- **y** (*pd.Series, np.ndarray*) – Future target of shape [n_samples].
- **X_train** (*pd.DataFrame, np.ndarray*) – Data the pipeline was trained on of shape [n_samples_train, n_features].
- **y_train** (*pd.Series, np.ndarray*) – Targets used to train the pipeline of shape [n_samples_train].
- **objective** (*ObjectiveBase, str, None*) – Objective used to threshold predicted probabilities, optional.

**Returns** Estimated labels.

**Return type** pd.Series

**Raises** **ValueError** – If final component is not an Estimator.

**predict_proba**(*self*, *X*, *X_train=None*, *y_train=None*)

Predict on future data where the target is unknown.

**Parameters**

- **X** (*pd.DataFrame or np.ndarray*) – Future data of shape [n_samples, n_features].
- **X_train** (*pd.DataFrame, np.ndarray*) – Data the pipeline was trained on of shape [n_samples_train, n_features].

- **y_train** (*pd.Series, np.ndarray*) – Targets used to train the pipeline of shape [n_samples_train].

**Returns** Estimated probabilities.

**Return type** pd.Series

**Raises** **ValueError** – If final component is not an Estimator.

**predict_proba_in_sample**(*self*, *X_holdout*, *y_holdout*, *X_train*, *y_train*)

Predict on future data where the target is known, e.g. cross validation.

**Parameters**

- **X_holdout** (*pd.DataFrame or np.ndarray*) – Future data of shape [n_samples, n_features].
- **y_holdout** (*pd.Series, np.ndarray*) – Future target of shape [n_samples].
- **X_train** (*pd.DataFrame, np.ndarray*) – Data the pipeline was trained on of shape [n_samples_train, n_features].
- **y_train** (*pd.Series, np.ndarray*) – Targets used to train the pipeline of shape [n_samples_train].

**Returns** Estimated probabilities.

**Return type** pd.Series

**Raises** **ValueError** – If the final component is not an Estimator.

**save**(*self*, *file_path*, *pickle_protocol=cloudpickle.DEFAULT_PROTOCOL*)

Saves pipeline at file path.

**Parameters**

- **file_path** (*str*) – Location to save file.
- **pickle_protocol** (*int*) – The pickle data stream format.

**score**(*self*, *X*, *y*, *objectives*, *X_train=None*, *y_train=None*)

Evaluate model performance on current and additional objectives.

**Parameters**

- **X** (*pd.DataFrame or np.ndarray*) – Data of shape [n_samples, n_features].
- **y** (*pd.Series*) – True labels of length [n_samples].
- **objectives** (*list*) – Non-empty list of objectives to score on.
- **X_train** (*pd.DataFrame, np.ndarray*) – Data the pipeline was trained on of shape [n_samples_train, n_features].
- **y_train** (*pd.Series, np.ndarray*) – Targets used to train the pipeline of shape [n_samples_train].

**Returns** Ordered dictionary of objective scores.

**Return type** dict

**property summary**(*self*)

A short summary of the pipeline structure, describing the list of components used.

Example: Logistic Regression Classifier w/ Simple Imputer + One Hot Encoder

**Returns** A string describing the pipeline structure.

**transform**(*self*, *X*, *y=None*)

Transform the input.

**Parameters**

- **X** (*pd.DataFrame, or np.ndarray*) – Data of shape [n_samples, n_features].
- **y** (*pd.Series*) – The target data of length [n_samples]. Defaults to None.

**Returns** Transformed output.

**Return type** pd.DataFrame

**transform_all_but_final**(*self*, *X*, *y=None*, *X_train=None*, *y_train=None*, *calculating_residuals=False*)

Transforms the data by applying all pre-processing components.

**Parameters**

- **X** (*pd.DataFrame*) – Input data to the pipeline to transform.
- **y** (*pd.Series*) – Targets corresponding to the pipeline targets.
- **X_train** (*pd.DataFrame*) – Training data used to generate generates from past observations.
- **y_train** (*pd.Series*) – Training targets used to generate features from past observations.
- **calculating_residuals** (*bool*) – Whether we're calling predict_in_sample to calculate the residuals. This means the X and y arguments are not future data, but actually the train data.

**Returns** New transformed features.

**Return type** pd.DataFrame

## time_series_pipeline_base

Pipeline base class for time-series problems.

### Module Contents

### Classes Summary

| | |
|---|---|
| *TimeSeriesPipelineBase* | Pipeline base class for time series problems. |

### Contents

**class** evalml.pipelines.time_series_pipeline_base.**TimeSeriesPipelineBase**(*component_graph*, *parameters=None*, *custom_name=None*, *random_seed=0*)

Pipeline base class for time series problems.

**Parameters**

- **component_graph** (*ComponentGraph, list, dict*) – ComponentGraph instance, list of components in order, or dictionary of components. Accepts strings or ComponentBase subclasses in the list. Note that when duplicate components are specified in a list, the duplicate component names will be modified with the component's index in the list. For example, the component graph [Imputer, One Hot Encoder, Imputer, Logistic Regression Classifier] will have names ["Imputer", "One Hot Encoder", "Imputer_2", "Logistic Regression Classifier"]
- **parameters** (*dict*) – Dictionary with component names as keys and dictionary of that component's parameters as values. An empty dictionary {} implies using all default values for component parameters. Pipeline-level parameters such as time_index, gap, and max_delay must be specified with the "pipeline" key. For example: Pipeline(parameters={"pipeline": {"time_index": "Date", "max_delay": 4, "gap": 2}}).
- **random_seed** (*int*) – Seed for the random number generator. Defaults to 0.

**Attributes**

| | |
|---|---|
| **problem_type** | None |

**Methods**

| | |
|---|---|
| *can_tune_threshold_with_objective* | Determine whether the threshold of a binary classification pipeline can be tuned. |
| *clone* | Constructs a new pipeline with the same components, parameters, and random seed. |
| *create_objectives* | Create objective instances from a list of strings or objective classes. |
| *custom_name* | Custom name of the pipeline. |
| *dates_needed_for_prediction* | Return dates needed to forecast the given date in the future. |
| *dates_needed_for_prediction_range* | Return dates needed to forecast the given date in the future. |
| *describe* | Outputs pipeline details including component parameters. |
| *feature_importance* | Importance associated with each feature. Features dropped by the feature selection are excluded. |
| *fit* | Build a model. |
| *fit_transform* | Fit and transform all components in the component graph, if all components are Transformers. |
| *get_component* | Returns component by name. |
| *get_hyperparameter_ranges* | Returns hyperparameter ranges from all components as a dictionary. |
| *graph* | Generate an image representing the pipeline graph. |
| *graph_dict* | Generates a dictionary with nodes consisting of the component names and parameters, and edges detailing component relationships. This dictionary is JSON serializable in most cases. |
| *graph_feature_importance* | Generate a bar graph of the pipeline's feature importance. |
| *inverse_transform* | Apply component inverse_transform methods to estimator predictions in reverse order. |
| *load* | Loads pipeline at file path. |
| *model_family* | Returns model family of this pipeline. |
| *name* | Name of the pipeline. |
| *new* | Constructs a new instance of the pipeline with the same component graph but with a different set of parameters. Not to be confused with python's __new__ method. |
| *parameters* | Parameter dictionary for this pipeline. |
| *predict* | Predict on future data where target is not known. |
| *predict_in_sample* | Predict on future data where the target is known, e.g. cross validation. |
| *save* | Saves pipeline at file path. |
| *score* | Evaluate model performance on current and additional objectives. |
| *summary* | A short summary of the pipeline structure, describing the list of components used. |
| *transform* | Transform the input. |
| *transform_all_but_final* | Transforms the data by applying all pre-processing components. |

**can_tune_threshold_with_objective**(*self*, *objective*)

Determine whether the threshold of a binary classification pipeline can be tuned.

**Parameters** **objective** (*ObjectiveBase*) – Primary AutoMLSearch objective.

**Returns** True if the pipeline threshold can be tuned.

**Return type** bool

**clone**(*self*)

Constructs a new pipeline with the same components, parameters, and random seed.

**Returns** A new instance of this pipeline with identical components, parameters, and random seed.

**static create_objectives**(*objectives*)

Create objective instances from a list of strings or objective classes.

**property custom_name**(*self*)

Custom name of the pipeline.

**dates_needed_for_prediction**(*self*, *date*)

Return dates needed to forecast the given date in the future.

**Parameters** **date** (*pd.Timestamp*) – Date to forecast in the future.

**Returns** Range of dates needed to forecast the given date.

**Return type** dates_needed (tuple(pd.Timestamp))

**dates_needed_for_prediction_range**(*self*, *start_date*, *end_date*)

Return dates needed to forecast the given date in the future.

**Parameters**

- **start_date** (*pd.Timestamp*) – Start date of range to forecast in the future.
- **end_date** (*pd.Timestamp*) – End date of range to forecast in the future.

**Returns** Range of dates needed to forecast the given date.

**Return type** dates_needed (tuple(pd.Timestamp))

**Raises** **ValueError** – If start_date doesn't come before end_date

**describe**(*self*, *return_dict=False*)

Outputs pipeline details including component parameters.

**Parameters** **return_dict** (*bool*) – If True, return dictionary of information about pipeline. Defaults to False.

**Returns** Dictionary of all component parameters if return_dict is True, else None.

**Return type** dict

**property feature_importance**(*self*)

Importance associated with each feature. Features dropped by the feature selection are excluded.

**Returns** Feature names and their corresponding importance

**Return type** pd.DataFrame

**abstract fit**(*self*, *X*, *y*)

Build a model.

**Parameters**

- **X** (*pd.DataFrame or np.ndarray*) – The input training data of shape [n_samples, n_features].

- **y** (*pd.Series, np.ndarray*) – The target training data of length [n_samples].

**Returns** self

**fit_transform**(*self*, *X*, *y*)

Fit and transform all components in the component graph, if all components are Transformers.

**Parameters**

- **X** (*pd.DataFrame*) – Input features of shape [n_samples, n_features].
- **y** (*pd.Series*) – The target data of length [n_samples].

**Returns** Transformed output.

**Return type** pd.DataFrame

**Raises** **ValueError** – If final component is an Estimator.

**get_component**(*self*, *name*)

Returns component by name.

**Parameters** **name** (*str*) – Name of component.

**Returns** Component to return

**Return type** Component

**get_hyperparameter_ranges**(*self*, *custom_hyperparameters*)

Returns hyperparameter ranges from all components as a dictionary.

**Parameters** **custom_hyperparameters** (*dict*) – Custom hyperparameters for the pipeline.

**Returns** Dictionary of hyperparameter ranges for each component in the pipeline.

**Return type** dict

**graph**(*self*, *filepath=None*)

Generate an image representing the pipeline graph.

**Parameters** **filepath** (*str, optional*) – Path to where the graph should be saved. If set to None (as by default), the graph will not be saved.

**Returns** Graph object that can be directly displayed in Jupyter notebooks.

**Return type** graphviz.Digraph

**Raises**

- **RuntimeError** – If graphviz is not installed.
- **ValueError** – If path is not writeable.

**graph_dict**(*self*)

Generates a dictionary with nodes consisting of the component names and parameters, and edges detailing component relationships. This dictionary is JSON serializable in most cases.

x_edges specifies from which component feature data is being passed. y_edges specifies from which component target data is being passed. This can be used to build graphs across a variety of visualization tools. Template: {"Nodes": {"component_name": {"Name": class_name, "Parameters": parameters_attributes}, ...}}, "x_edges": [[from_component_name, to_component_name], [from_component_name, to_component_name], ...], "y_edges": [[from_component_name, to_component_name], [from_component_name, to_component_name], ...]}

**Returns** A dictionary representing the DAG structure.

**Return type** dag_dict (dict)

**graph_feature_importance**(*self*, *importance_threshold=0*)

Generate a bar graph of the pipeline's feature importance.

**Parameters** **importance_threshold** (*float, optional*) – If provided, graph features with a permutation importance whose absolute value is larger than importance_threshold. Defaults to zero.

**Returns** A bar graph showing features and their corresponding importance.

**Return type** plotly.Figure

**Raises** **ValueError** – If importance threshold is not valid.

**inverse_transform**(*self*, *y*)

Apply component inverse_transform methods to estimator predictions in reverse order.

Components that implement inverse_transform are PolynomialDecomposer, LogTransformer, LabelEncoder (tbd).

**Parameters** **y** (*pd.Series*) – Final component features.

**Returns** The inverse transform of the target.

**Return type** pd.Series

**static load**(*file_path*)

Loads pipeline at file path.

**Parameters** **file_path** (*str*) – Location to load file.

**Returns** PipelineBase object

**property model_family**(*self*)

Returns model family of this pipeline.

**property name**(*self*)

Name of the pipeline.

**new**(*self*, *parameters*, *random_seed=0*)

Constructs a new instance of the pipeline with the same component graph but with a different set of parameters. Not to be confused with python's __new__ method.

**Parameters**

- **parameters** (*dict*) – Dictionary with component names as keys and dictionary of that component's parameters as values. An empty dictionary or None implies using all default values for component parameters. Defaults to None.
- **random_seed** (*int*) – Seed for the random number generator. Defaults to 0.

**Returns** A new instance of this pipeline with identical components.

**property parameters**(*self*)

Parameter dictionary for this pipeline.

**Returns** Dictionary of all component parameters.

**Return type** dict

**predict**(*self*, *X*, *objective=None*, *X_train=None*, *y_train=None*)

Predict on future data where target is not known.

**Parameters**

- **X** (*pd.DataFrame, or np.ndarray*) – Data of shape [n_samples, n_features].
- **objective** (*Object or string*) – The objective to use to make predictions.
- **X_train** (*pd.DataFrame or np.ndarray or None*) – Training data.
- **y_train** (*pd.Series or None*) – Training labels.

**Raises** **ValueError** – If X_train and/or y_train are None or if final component is not an Estimator.

**Returns** Predictions.

**predict_in_sample**(*self*, *X*, *y*, *X_train*, *y_train*, *objective=None*, *calculating_residuals=False*)

Predict on future data where the target is known, e.g. cross validation.

**Parameters**

- **X** (*pd.DataFrame or np.ndarray*) – Future data of shape [n_samples, n_features]
- **y** (*pd.Series, np.ndarray*) – Future target of shape [n_samples]
- **X_train** (*pd.DataFrame, np.ndarray*) – Data the pipeline was trained on of shape [n_samples_train, n_feautures]
- **y_train** (*pd.Series, np.ndarray*) – Targets used to train the pipeline of shape [n_samples_train]
- **objective** (*ObjectiveBase, str, None*) – Objective used to threshold predicted probabilities, optional.
- **calculating_residuals** (*bool*) – Whether we're calling predict_in_sample to calculate the residuals. This means the X and y arguments are not future data, but actually the train data.

**Returns** Estimated labels.

**Return type** pd.Series

**Raises** **ValueError** – If final component is not an Estimator.

**save**(*self*, *file_path*, *pickle_protocol=cloudpickle.DEFAULT_PROTOCOL*)

Saves pipeline at file path.

**Parameters**

- **file_path** (*str*) – Location to save file.
- **pickle_protocol** (*int*) – The pickle data stream format.

**abstract score**(*self*, *X*, *y*, *objectives*, *X_train=None*, *y_train=None*)

Evaluate model performance on current and additional objectives.

**Parameters**

- **X** (*pd.DataFrame or np.ndarray*) – Data of shape [n_samples, n_features].
- **y** (*pd.Series, np.ndarray*) – True labels of length [n_samples].
- **objectives** (*list*) – Non-empty list of objectives to score on.

- **X_train** (*pd.DataFrame or np.ndarray or None*) – Training data. Ignored. Only used for time series.
- **y_train** (*pd.Series or None*) – Training labels. Ignored. Only used for time series.

**Returns** Ordered dictionary of objective scores.

**Return type** dict

**property summary**(*self*)

A short summary of the pipeline structure, describing the list of components used.

Example: Logistic Regression Classifier w/ Simple Imputer + One Hot Encoder

**Returns** A string describing the pipeline structure.

**transform**(*self*, *X*, *y=None*)

Transform the input.

**Parameters**

- **X** (*pd.DataFrame, or np.ndarray*) – Data of shape [n_samples, n_features].
- **y** (*pd.Series*) – The target data of length [n_samples]. Defaults to None.

**Returns** Transformed output.

**Return type** pd.DataFrame

**transform_all_but_final**(*self*, *X*, *y=None*, *X_train=None*, *y_train=None*, *calculating_residuals=False*)

Transforms the data by applying all pre-processing components.

**Parameters**

- **X** (*pd.DataFrame*) – Input data to the pipeline to transform.
- **y** (*pd.Series*) – Targets corresponding to the pipeline targets.
- **X_train** (*pd.DataFrame*) – Training data used to generate generates from past observations.
- **y_train** (*pd.Series*) – Training targets used to generate features from past observations.
- **calculating_residuals** (*bool*) – Whether we're calling predict_in_sample to calculate the residuals. This means the X and y arguments are not future data, but actually the train data.

**Returns** New transformed features.

**Return type** pd.DataFrame

### time_series_regression_pipeline

Pipeline base class for time series regression problems.

### Module Contents

### Classes Summary

| | |
|---|---|
| *TimeSeriesRegressionPipeline* | Pipeline base class for time series regression problems. |

### Contents

**class** evalml.pipelines.time_series_regression_pipeline.**TimeSeriesRegressionPipeline**(*component_graph, parameters=None, custom_name=None, random_seed=0*)

Pipeline base class for time series regression problems.

**Parameters**

- **component_graph** (*ComponentGraph, list, dict*) – ComponentGraph instance, list of components in order, or dictionary of components. Accepts strings or ComponentBase subclasses in the list. Note that when duplicate components are specified in a list, the duplicate component names will be modified with the component's index in the list. For example, the component graph [Imputer, One Hot Encoder, Imputer, Logistic Regression Classifier] will have names ["Imputer", "One Hot Encoder", "Imputer_2", "Logistic Regression Classifier"]
- **parameters** (*dict*) – Dictionary with component names as keys and dictionary of that component's parameters as values. An empty dictionary {} implies using all default values for component parameters. Pipeline-level parameters such as time_index, gap, and max_delay must be specified with the "pipeline" key. For example: Pipeline(parameters={"pipeline": {"time_index": "Date", "max_delay": 4, "gap": 2}}).
- **random_seed** (*int*) – Seed for the random number generator. Defaults to 0.

### Example

```
>>> pipeline = TimeSeriesRegressionPipeline(component_graph=["Simple Imputer",
↪"Linear Regressor"],
...                                          parameters={"Simple␣
↪Imputer": {"impute_strategy": "mean"},
...                                                      "pipeline": {
↪"gap": 1, "max_delay": 1, "forecast_horizon": 1, "time_index": "date"}},
...                                          custom_name="My␣
↪TimeSeriesRegression Pipeline")
...
>>> assert pipeline.custom_name == "My TimeSeriesRegression Pipeline"
>>> assert pipeline.component_graph.component_dict.keys() == {'Simple Imputer',
↪'Linear Regressor'}
```

The pipeline parameters will be chosen from the default parameters for every component, unless specific parameters were passed in as they were above.

```
>>> assert pipeline.parameters == {
...     'Simple Imputer': {'impute_strategy': 'mean', 'fill_value': None},
...     'Linear Regressor': {'fit_intercept': True, 'n_jobs': -1},
...     'pipeline': {'gap': 1, 'max_delay': 1, 'forecast_horizon': 1, 'time_index':
↪"date"}}
```

**Attributes**

| | |
|---|---|
| **NO_PREDS_PIPELINE_TYPES** | [ESTIMATOR, TIME_SERIES_REGRESSION] |
| **problem_type** | None |

**Methods**

| | |
|---|---|
| *can_tune_threshold_with_objective* | Determine whether the threshold of a binary classification pipeline can be tuned. |
| *clone* | Constructs a new pipeline with the same components, parameters, and random seed. |
| *create_objectives* | Create objective instances from a list of strings or objective classes. |
| *custom_name* | Custom name of the pipeline. |
| *dates_needed_for_prediction* | Return dates needed to forecast the given date in the future. |
| *dates_needed_for_prediction_range* | Return dates needed to forecast the given date in the future. |
| *describe* | Outputs pipeline details including component parameters. |
| *feature_importance* | Importance associated with each feature. Features dropped by the feature selection are excluded. |
| *fit* | Fit a time series pipeline. |
| *fit_transform* | Fit and transform all components in the component graph, if all components are Transformers. |
| *get_component* | Returns component by name. |
| *get_forecast_period* | Generates all possible forecasting time points based on latest data point in X. |
| *get_forecast_predictions* | Generates all possible forecasting predictions based on last period of X. |
| *get_hyperparameter_ranges* | Returns hyperparameter ranges from all components as a dictionary. |
| *get_prediction_intervals* | Find the prediction intervals using the fitted regressor. |
| *graph* | Generate an image representing the pipeline graph. |
| *graph_dict* | Generates a dictionary with nodes consisting of the component names and parameters, and edges detailing component relationships. This dictionary is JSON serializable in most cases. |
| *graph_feature_importance* | Generate a bar graph of the pipeline's feature importance. |

continues on next page

Table 10 – continued from previous page

| | |
|---|---|
| *inverse_transform* | Apply component inverse_transform methods to estimator predictions in reverse order. |
| *load* | Loads pipeline at file path. |
| *model_family* | Returns model family of this pipeline. |
| *name* | Name of the pipeline. |
| *new* | Constructs a new instance of the pipeline with the same component graph but with a different set of parameters. Not to be confused with python's __new__ method. |
| *parameters* | Parameter dictionary for this pipeline. |
| *predict* | Predict on future data where target is not known. |
| *predict_in_sample* | Predict on future data where the target is known, e.g. cross validation. |
| *save* | Saves pipeline at file path. |
| *score* | Evaluate model performance on current and additional objectives. |
| *summary* | A short summary of the pipeline structure, describing the list of components used. |
| *transform* | Transform the input. |
| *transform_all_but_final* | Transforms the data by applying all pre-processing components. |

**can_tune_threshold_with_objective**(*self*, *objective*)

Determine whether the threshold of a binary classification pipeline can be tuned.

**Parameters** **objective** (*ObjectiveBase*) – Primary AutoMLSearch objective.

**Returns** True if the pipeline threshold can be tuned.

**Return type** bool

**clone**(*self*)

Constructs a new pipeline with the same components, parameters, and random seed.

**Returns** A new instance of this pipeline with identical components, parameters, and random seed.

**static create_objectives**(*objectives*)

Create objective instances from a list of strings or objective classes.

**property custom_name**(*self*)

Custom name of the pipeline.

**dates_needed_for_prediction**(*self*, *date*)

Return dates needed to forecast the given date in the future.

**Parameters** **date** (*pd.Timestamp*) – Date to forecast in the future.

**Returns** Range of dates needed to forecast the given date.

**Return type** dates_needed (tuple(pd.Timestamp))

**dates_needed_for_prediction_range**(*self*, *start_date*, *end_date*)

Return dates needed to forecast the given date in the future.

**Parameters**

- **start_date** (*pd.Timestamp*) – Start date of range to forecast in the future.

- **end_date** (*pd.Timestamp*) – End date of range to forecast in the future.

**Returns** Range of dates needed to forecast the given date.

**Return type** dates_needed (tuple(pd.Timestamp))

**Raises** **ValueError** – If start_date doesn't come before end_date

**describe**(*self*, *return_dict=False*)

Outputs pipeline details including component parameters.

**Parameters** **return_dict** (*bool*) – If True, return dictionary of information about pipeline. Defaults to False.

**Returns** Dictionary of all component parameters if return_dict is True, else None.

**Return type** dict

**property feature_importance**(*self*)

Importance associated with each feature. Features dropped by the feature selection are excluded.

**Returns** Feature names and their corresponding importance

**Return type** pd.DataFrame

**fit**(*self*, *X*, *y*)

Fit a time series pipeline.

**Parameters**

- **X** (*pd.DataFrame or np.ndarray*) – The input training data of shape [n_samples, n_features].
- **y** (*pd.Series, np.ndarray*) – The target training targets of length [n_samples].

**Returns** self

**Raises** **ValueError** – If the target is not numeric.

**fit_transform**(*self*, *X*, *y*)

Fit and transform all components in the component graph, if all components are Transformers.

**Parameters**

- **X** (*pd.DataFrame*) – Input features of shape [n_samples, n_features].
- **y** (*pd.Series*) – The target data of length [n_samples].

**Returns** Transformed output.

**Return type** pd.DataFrame

**Raises** **ValueError** – If final component is an Estimator.

**get_component**(*self*, *name*)

Returns component by name.

**Parameters** **name** (*str*) – Name of component.

**Returns** Component to return

**Return type** Component

**get_forecast_period**(*self*, *X*)

Generates all possible forecasting time points based on latest data point in X.

**Parameters** **X** (*pd.DataFrame, np.ndarray*) – Data the pipeline was trained on of shape [n_samples_train, n_feautures].

**Raises** **ValueError** – If pipeline is not trained.

**Returns** Datetime periods out to *forecast_horizon* + *gap*.

**Return type** pd.Series

### Example

```
>>> X = pd.DataFrame({'date': pd.date_range(start='1-1-2022', periods=10, freq=
↪'D'), 'feature': range(10, 20)})
>>> y = pd.Series(range(0, 10), name='target')
>>> gap = 1
>>> forecast_horizon = 2
>>> pipeline = TimeSeriesRegressionPipeline(component_graph=["Linear Regressor
↪"],
...                                          parameters={"Simple Imputer": {
↪"impute_strategy": "mean"},
...                                                      "pipeline": {"gap": gap,
↪ "max_delay": 1, "forecast_horizon": forecast_horizon, "time_index": "date"}},
...                                         )
>>> pipeline.fit(X, y)
pipeline = TimeSeriesRegressionPipeline(component_graph={'Linear Regressor': [
↪'Linear Regressor', 'X', 'y']}, parameters={'Linear Regressor':{'fit_intercept
↪': True, 'n_jobs': -1}, 'pipeline':{'gap': 1, 'max_delay': 1, 'forecast_
↪horizon': 2, 'time_index': 'date'}}, random_seed=0)
>>> dates = pipeline.get_forecast_period(X)
>>> expected = pd.Series(pd.date_range(start='2022-01-11', periods=(gap +␣
↪forecast_horizon), freq='D'), name='date', index=[10, 11, 12])
>>> assert dates.equals(expected)
```

**get_forecast_predictions**(*self*, *X*, *y*)

Generates all possible forecasting predictions based on last period of X.

**Parameters**

- **X** (*pd.DataFrame, np.ndarray*) – Data the pipeline was trained on of shape [n_samples_train, n_feautures].
- **y** (*pd.Series, np.ndarray*) – Targets used to train the pipeline of shape [n_samples_train].

**Returns** Predictions out to *forecast_horizon* + *gap* periods.

**get_hyperparameter_ranges**(*self*, *custom_hyperparameters*)

Returns hyperparameter ranges from all components as a dictionary.

**Parameters** **custom_hyperparameters** (*dict*) – Custom hyperparameters for the pipeline.

**Returns** Dictionary of hyperparameter ranges for each component in the pipeline.

**Return type** dict

**get_prediction_intervals**(*self*, *X*, *y=None*, *X_train=None*, *y_train=None*, *coverage=None*)

Find the prediction intervals using the fitted regressor.

This function takes the predictions of the fitted estimator and calculates the rolling standard deviation across all predictions using a window size of 5. The lower and upper predictions are determined by taking the percent point (quantile) function of the lower tail probability at each bound multiplied by the rolling standard deviation.

Certain estimators (Extra Trees Estimator, XGBoost Estimator, Prophet Estimator, ARIMA, and Exponential Smoothing estimator) utilize a different methodology to calculate prediction intervals. See the docs for these estimators to learn more.

**Parameters**

- **X** (*pd.DataFrame*) – Data of shape [n_samples, n_features].
- **y** (*pd.Series*) – Target data.
- **X_train** (*pd.DataFrame, np.ndarray*) – Data the pipeline was trained on of shape [n_samples_train, n_features].
- **y_train** (*pd.Series, np.ndarray*) – Targets used to train the pipeline of shape [n_samples_train].
- **coverage** (*list[float]*) – A list of floats between the values 0 and 1 that the upper and lower bounds of the prediction interval should be calculated for.

**Returns** Prediction intervals, keys are in the format {coverage}_lower or {coverage}_upper.

**Return type** dict

**Raises** **MethodPropertyNotFoundError** – If the estimator does not support Time Series Regression as a problem type.

**graph**(*self*, *filepath=None*)

Generate an image representing the pipeline graph.

**Parameters** **filepath** (*str, optional*) – Path to where the graph should be saved. If set to None (as by default), the graph will not be saved.

**Returns** Graph object that can be directly displayed in Jupyter notebooks.

**Return type** graphviz.Digraph

**Raises**

- **RuntimeError** – If graphviz is not installed.
- **ValueError** – If path is not writeable.

**graph_dict**(*self*)

Generates a dictionary with nodes consisting of the component names and parameters, and edges detailing component relationships. This dictionary is JSON serializable in most cases.

x_edges specifies from which component feature data is being passed. y_edges specifies from which component target data is being passed. This can be used to build graphs across a variety of visualization tools. Template: {"Nodes": {"component_name": {"Name": class_name, "Parameters": parameters_attributes}, ...}}, "x_edges": [[from_component_name, to_component_name], [from_component_name, to_component_name], ...], "y_edges": [[from_component_name, to_component_name], [from_component_name, to_component_name], ...]}

**Returns** A dictionary representing the DAG structure.

**Return type** dag_dict (dict)

**graph_feature_importance**(*self*, *importance_threshold=0*)

Generate a bar graph of the pipeline's feature importance.

**Parameters** **importance_threshold** (*float, optional*) – If provided, graph features with a permutation importance whose absolute value is larger than importance_threshold. Defaults to zero.

**Returns** A bar graph showing features and their corresponding importance.

**Return type** plotly.Figure

**Raises** **ValueError** – If importance threshold is not valid.

**inverse_transform**(*self*, *y*)

Apply component inverse_transform methods to estimator predictions in reverse order.

Components that implement inverse_transform are PolynomialDecomposer, LogTransformer, LabelEncoder (tbd).

**Parameters** **y** (*pd.Series*) – Final component features.

**Returns** The inverse transform of the target.

**Return type** pd.Series

**static load**(*file_path*)

Loads pipeline at file path.

**Parameters** **file_path** (*str*) – Location to load file.

**Returns** PipelineBase object

**property model_family**(*self*)

Returns model family of this pipeline.

**property name**(*self*)

Name of the pipeline.

**new**(*self*, *parameters*, *random_seed=0*)

Constructs a new instance of the pipeline with the same component graph but with a different set of parameters. Not to be confused with python's __new__ method.

**Parameters**

- **parameters** (*dict*) – Dictionary with component names as keys and dictionary of that component's parameters as values. An empty dictionary or None implies using all default values for component parameters. Defaults to None.
- **random_seed** (*int*) – Seed for the random number generator. Defaults to 0.

**Returns** A new instance of this pipeline with identical components.

**property parameters**(*self*)

Parameter dictionary for this pipeline.

**Returns** Dictionary of all component parameters.

**Return type** dict

**predict**(*self*, *X*, *objective=None*, *X_train=None*, *y_train=None*)

Predict on future data where target is not known.

**Parameters**

- **X** (*pd.DataFrame, or np.ndarray*) – Data of shape [n_samples, n_features].

- **objective** (*Object or string*) – The objective to use to make predictions.
- **X_train** (*pd.DataFrame or np.ndarray or None*) – Training data.
- **y_train** (*pd.Series or None*) – Training labels.

**Raises** **ValueError** – If X_train and/or y_train are None or if final component is not an Estimator.

**Returns** Predictions.

**predict_in_sample**(*self*, *X*, *y*, *X_train*, *y_train*, *objective=None*, *calculating_residuals=False*)

Predict on future data where the target is known, e.g. cross validation.

**Parameters**

- **X** (*pd.DataFrame or np.ndarray*) – Future data of shape [n_samples, n_features]
- **y** (*pd.Series, np.ndarray*) – Future target of shape [n_samples]
- **X_train** (*pd.DataFrame, np.ndarray*) – Data the pipeline was trained on of shape [n_samples_train, n_feautures]
- **y_train** (*pd.Series, np.ndarray*) – Targets used to train the pipeline of shape [n_samples_train]
- **objective** (*ObjectiveBase, str, None*) – Objective used to threshold predicted probabilities, optional.
- **calculating_residuals** (*bool*) – Whether we're calling predict_in_sample to calculate the residuals. This means the X and y arguments are not future data, but actually the train data.

**Returns** Estimated labels.

**Return type** pd.Series

**Raises** **ValueError** – If final component is not an Estimator.

**save**(*self*, *file_path*, *pickle_protocol=cloudpickle.DEFAULT_PROTOCOL*)

Saves pipeline at file path.

**Parameters**

- **file_path** (*str*) – Location to save file.
- **pickle_protocol** (*int*) – The pickle data stream format.

**score**(*self*, *X*, *y*, *objectives*, *X_train=None*, *y_train=None*)

Evaluate model performance on current and additional objectives.

**Parameters**

- **X** (*pd.DataFrame or np.ndarray*) – Data of shape [n_samples, n_features].
- **y** (*pd.Series*) – True labels of length [n_samples].
- **objectives** (*list*) – Non-empty list of objectives to score on.
- **X_train** (*pd.DataFrame, np.ndarray*) – Data the pipeline was trained on of shape [n_samples_train, n_feautures].
- **y_train** (*pd.Series, np.ndarray*) – Targets used to train the pipeline of shape [n_samples_train].

**Returns** Ordered dictionary of objective scores.

**Return type** dict

**property summary**(*self*)

A short summary of the pipeline structure, describing the list of components used.

Example: Logistic Regression Classifier w/ Simple Imputer + One Hot Encoder

**Returns** A string describing the pipeline structure.

**transform**(*self*, *X*, *y=None*)

Transform the input.

**Parameters**

- **X** (*pd.DataFrame, or np.ndarray*) – Data of shape [n_samples, n_features].
- **y** (*pd.Series*) – The target data of length [n_samples]. Defaults to None.

**Returns** Transformed output.

**Return type** pd.DataFrame

**transform_all_but_final**(*self*, *X*, *y=None*, *X_train=None*, *y_train=None*, *calculating_residuals=False*)

Transforms the data by applying all pre-processing components.

**Parameters**

- **X** (*pd.DataFrame*) – Input data to the pipeline to transform.
- **y** (*pd.Series*) – Targets corresponding to the pipeline targets.
- **X_train** (*pd.DataFrame*) – Training data used to generate generates from past observations.
- **y_train** (*pd.Series*) – Training targets used to generate features from past observations.
- **calculating_residuals** (*bool*) – Whether we're calling predict_in_sample to calculate the residuals. This means the X and y arguments are not future data, but actually the train data.

**Returns** New transformed features.

**Return type** pd.DataFrame

### utils

Utility methods for EvalML pipelines.

### Module Contents

### Functions

| | |
|---|---|
| *generate_pipeline_code* | Creates and returns a string that contains the Python imports and code required for running the EvalML pipeline. |
| *generate_pipeline_example* | Creates and returns a string that contains the Python imports and code required for running the EvalML pipeline. |
| *get_actions_from_option_defaults* | Returns a list of actions based on the defaults parameters of each option in the input DataCheckActionOption list. |
| *make_pipeline* | Given input data, target data, an estimator class and the problem type, generates a pipeline class with a preprocessing chain which was recommended based on the inputs. The pipeline will be a subclass of the appropriate pipeline base class for the specified problem_type. |
| *make_pipeline_from_actions* | Creates a pipeline of components to address the input DataCheckAction list. |
| *make_pipeline_from_data_check_output* | Creates a pipeline of components to address warnings and errors output from running data checks. Uses all default suggestions. |
| *make_timeseries_baseline_pipeline* | Make a baseline pipeline for time series regression problems. |
| *rows_of_interest* | Get the row indices of the data that are closest to the threshold. Works only for binary classification problems and pipelines. |

### Attributes Summary

| | |
|---|---|
| *DECOMPOSER_PERIOD_CAP* | |

### Contents

evalml.pipelines.utils.**DECOMPOSER_PERIOD_CAP = 1000**

evalml.pipelines.utils.**generate_pipeline_code**(*element*, *features_path=None*)

Creates and returns a string that contains the Python imports and code required for running the EvalML pipeline.

**Parameters**

- **element** (*pipeline instance*) – The instance of the pipeline to generate string Python code.
- **features_path** (*str*) – path to features json created from featuretools.save_features(). Defaults to None.

**Returns** String representation of Python code that can be run separately in order to recreate the pipeline instance. Does not include code for custom component implementation.

**Return type** str

**Raises**

- **ValueError** – If element is not a pipeline, or if the pipeline is nonlinear.
- **ValueError** – If features in *features_path* do not match the features on the pipeline.

evalml.pipelines.utils.**generate_pipeline_example**(*pipeline*, *path_to_train*, *path_to_holdout*, *target*, *path_to_features=None*, *path_to_mapping=''*, *output_file_path=None*)

Creates and returns a string that contains the Python imports and code required for running the EvalML pipeline.

**Parameters**

- **pipeline** (*pipeline instance*) – The instance of the pipeline to generate string Python code.
- **path_to_train** (*str*) – path to training data.
- **path_to_holdout** (*str*) – path to holdout data.
- **target** (*str*) – target variable.
- **path_to_features** (*str*) – path to features json. Defaults to None.
- **path_to_mapping** (*str*) – path to mapping json. Defaults to None.
- **output_file_path** (*str*) – path to output python file. Defaults to None.

**Returns** String representation of Python code that can be run separately in order to recreate the pipeline instance. Does not include code for custom component implementation.

**Return type** str

evalml.pipelines.utils.**get_actions_from_option_defaults**(*action_options*)

Returns a list of actions based on the defaults parameters of each option in the input DataCheckActionOption list.

**Parameters** **action_options** (*list[DataCheckActionOption]*) – List of DataCheckActionOption objects

**Returns** List of actions based on the defaults parameters of each option in the input list.

**Return type** list[DataCheckAction]

evalml.pipelines.utils.**make_pipeline**(*X*, *y*, *estimator*, *problem_type*, *parameters=None*, *sampler_name=None*, *extra_components_before=None*, *extra_components_after=None*, *use_estimator=True*, *known_in_advance=None*, *features=False*, *exclude_featurizers=None*, *include_decomposer=True*)

Given input data, target data, an estimator class and the problem type, generates a pipeline class with a preprocessing chain which was recommended based on the inputs. The pipeline will be a subclass of the appropriate pipeline base class for the specified problem_type.

**Parameters**

- **X** (*pd.DataFrame*) – The input data of shape [n_samples, n_features].
- **y** (*pd.Series*) – The target data of length [n_samples].
- **estimator** (*Estimator*) – Estimator for pipeline.
- **problem_type** (*ProblemTypes or str*) – Problem type for pipeline to generate.
- **parameters** (*dict*) – Dictionary with component names as keys and dictionary of that component's parameters as values. An empty dictionary or None implies using all default values for component parameters.

- **sampler_name** (*str*) – The name of the sampler component to add to the pipeline. Only used in classification problems. Defaults to None
- **extra_components_before** (*list[ComponentBase]*) – List of extra components to be added before preprocessing components. Defaults to None.
- **extra_components_after** (*list[ComponentBase]*) – List of extra components to be added after preprocessing components. Defaults to None.
- **use_estimator** (*bool*) – Whether to add the provided estimator to the pipeline or not. Defaults to True.
- **known_in_advance** (*list[str], None*) – List of features that are known in advance.
- **features** (*bool*) – Whether to add a DFSTransformer component to this pipeline.
- **exclude_featurizers** (*list[str]*) – A list of featurizer components to exclude from the pipeline. Valid options are "DatetimeFeaturizer", "EmailFeaturizer", "URLFeaturizer", "NaturalLanguageFeaturizer", "TimeSeriesFeaturizer"
- **include_decomposer** (*bool*) – For time series regression problems, whether or not to include a decomposer in the generated pipeline. Defaults to True.

**Returns** PipelineBase instance with dynamically generated preprocessing components and specified estimator.

**Return type** PipelineBase object

**Raises** **ValueError** – If estimator is not valid for the given problem type, or sampling is not supported for the given problem type.

`evalml.pipelines.utils.`**make_pipeline_from_actions**(*problem_type*, *actions*, *problem_configuration=None*)

Creates a pipeline of components to address the input DataCheckAction list.

**Parameters**

- **problem_type** (*str or ProblemType*) – The problem type that the pipeline should address.
- **actions** (*list[DataCheckAction]*) – List of DataCheckAction objects used to create list of components
- **problem_configuration** (*dict*) – Required for time series problem types. Values should be passed in for time_index, gap, forecast_horizon, and max_delay.

**Returns** Pipeline which can be used to address data check actions.

**Return type** PipelineBase

`evalml.pipelines.utils.`**make_pipeline_from_data_check_output**(*problem_type*, *data_check_output*, *problem_configuration=None*)

Creates a pipeline of components to address warnings and errors output from running data checks. Uses all default suggestions.

**Parameters**

- **problem_type** (*str or ProblemType*) – The problem type.
- **data_check_output** (*dict*) – Output from calling `DataCheck.validate()`.
- **problem_configuration** (*dict*) – Required for time series problem types. Values should be passed in for time_index, gap, forecast_horizon, and max_delay.

**Returns** Pipeline which can be used to address data check outputs.

**Return type** PipelineBase

**Raises** **ValueError** – If problem_type is of type time series but an incorrect problem_configuration has been passed.

`evalml.pipelines.utils.make_timeseries_baseline_pipeline(problem_type, gap, forecast_horizon, time_index, exclude_featurizer=False)`

Make a baseline pipeline for time series regression problems.

**Parameters**

- **problem_type** – One of TIME_SERIES_REGRESSION, TIME_SERIES_MULTICLASS, TIME_SERIES_BINARY
- **gap** (*int*) – Non-negative gap parameter.
- **forecast_horizon** (*int*) – Positive forecast_horizon parameter.
- **time_index** (*str*) – Column name of time_index parameter.
- **exclude_featurizer** (*bool*) – Whether or not to exclude the TimeSeriesFeaturizer from the baseline graph. Defaults to False.

**Returns** TimeSeriesPipelineBase, a time series pipeline corresponding to the problem type.

`evalml.pipelines.utils.rows_of_interest(pipeline, X, y=None, threshold=None, epsilon=0.1, sort_values=True, types='all')`

Get the row indices of the data that are closest to the threshold. Works only for binary classification problems and pipelines.

**Parameters**

- **pipeline** (*PipelineBase*) – The fitted binary pipeline.
- **X** (*ww.DataTable, pd.DataFrame*) – The input features to predict on.
- **y** (*ww.DataColumn, pd.Series, None*) – The input target data, if available. Defaults to None.
- **threshold** (*float*) – The threshold value of interest to separate positive and negative predictions. If None, uses the pipeline threshold if set, else 0.5. Defaults to None.
- **epsilon** (*epsilon*) – The difference between the probability and the threshold that would make the row interesting for us. For instance, epsilon=0.1 and threhsold=0.5 would mean we consider all rows in [0.4, 0.6] to be of interest. Defaults to 0.1.
- **sort_values** (*bool*) – Whether to return the indices sorted by the distance from the threshold, such that the first values are closer to the threshold and the later values are further. Defaults to True.
- **types** (*str*) – The type of rows to keep and return. Can be one of [‘incorrect’, ‘correct’, ‘true_positive’, ‘true_negative’, ‘all’]. Defaults to ‘all’.

  ’incorrect’ - return only the rows where the predictions are incorrect. This means that, given the threshold and target y, keep only the rows which are labeled wrong. ‘correct’ - return only the rows where the predictions are correct. This means that, given the threshold and target y, keep only the rows which are correctly labeled. ‘true_positive’ - return only the rows which are positive, as given by the targets. ‘true_negative’ - return only the rows which are negative, as given by the targets. ‘all’ - return all rows. This is the only option available when there is no target data provided.

**Returns** The indices corresponding to the rows of interest.

**Raises**

- **ValueError** – If pipeline is not a fitted Binary Classification pipeline.
- **ValueError** – If types is invalid or y is not provided when types is not 'all'.
- **ValueError** – If the threshold is provided and is exclusive of [0, 1].

### Package Contents

### Classes Summary

| | |
|---|---|
| *ARIMARegressor* | Autoregressive Integrated Moving Average Model. The three parameters (p, d, q) are the AR order, the degree of differencing, and the MA order. More information here: https://www.statsmodels.org/devel/generated/statsmodels.tsa.arima.model.ARIMA.html. |
| *BinaryClassificationPipeline* | Pipeline subclass for all binary classification pipelines. |
| *CatBoostClassifier* | CatBoost Classifier, a classifier that uses gradient-boosting on decision trees. CatBoost is an open-source library and natively supports categorical features. |
| *CatBoostRegressor* | CatBoost Regressor, a regressor that uses gradient-boosting on decision trees. CatBoost is an open-source library and natively supports categorical features. |
| *ClassificationPipeline* | Pipeline subclass for all classification pipelines. |
| *ComponentGraph* | Component graph for a pipeline as a directed acyclic graph (DAG). |
| *DecisionTreeClassifier* | Decision Tree Classifier. |
| *DecisionTreeRegressor* | Decision Tree Regressor. |
| *DFSTransformer* | Featuretools DFS component that generates features for the input features. |
| *DropNaNRowsTransformer* | Transformer to drop rows with NaN values. |
| *ElasticNetClassifier* | Elastic Net Classifier. Uses Logistic Regression with elasticnet penalty as the base estimator. |
| *ElasticNetRegressor* | Elastic Net Regressor. |
| *Estimator* | A component that fits and predicts given data. |
| *ExponentialSmoothingRegressor* | Holt-Winters Exponential Smoothing Forecaster. |
| *ExtraTreesClassifier* | Extra Trees Classifier. |
| *ExtraTreesRegressor* | Extra Trees Regressor. |
| *FeatureSelector* | Selects top features based on importance weights. |
| *Imputer* | Imputes missing data according to a specified imputation strategy. |
| *KNeighborsClassifier* | K-Nearest Neighbors Classifier. |
| *LightGBMClassifier* | LightGBM Classifier. |
| *LightGBMRegressor* | LightGBM Regressor. |
| *LinearRegressor* | Linear Regressor. |
| *LogisticRegressionClassifier* | Logistic Regression Classifier. |
| *MulticlassClassificationPipeline* | Pipeline subclass for all multiclass classification pipelines. |
| *OneHotEncoder* | A transformer that encodes categorical features in a one-hot numeric array. |

continues on next page

Table 11 – continued from previous page

| | |
|---|---|
| *OrdinalEncoder* | A transformer that encodes ordinal features as an array of ordinal integers representing the relative order of categories. |
| *PerColumnImputer* | Imputes missing data according to a specified imputation strategy per column. |
| *PipelineBase* | Machine learning pipeline. |
| *ProphetRegressor* | Prophet is a procedure for forecasting time series data based on an additive model where non-linear trends are fit with yearly, weekly, and daily seasonality, plus holiday effects. It works best with time series that have strong seasonal effects and several seasons of historical data. Prophet is robust to missing data and shifts in the trend, and typically handles outliers well. |
| *RandomForestClassifier* | Random Forest Classifier. |
| *RandomForestRegressor* | Random Forest Regressor. |
| *RegressionPipeline* | Pipeline subclass for all regression pipelines. |
| *RFClassifierSelectFromModel* | Selects top features based on importance weights using a Random Forest classifier. |
| *RFRegressorSelectFromModel* | Selects top features based on importance weights using a Random Forest regressor. |
| *SimpleImputer* | Imputes missing data according to a specified imputation strategy. Natural language columns are ignored. |
| *StackedEnsembleBase* | Stacked Ensemble Base Class. |
| *StackedEnsembleClassifier* | Stacked Ensemble Classifier. |
| *StackedEnsembleRegressor* | Stacked Ensemble Regressor. |
| *StandardScaler* | A transformer that standardizes input features by removing the mean and scaling to unit variance. |
| *SVMClassifier* | Support Vector Machine Classifier. |
| *SVMRegressor* | Support Vector Machine Regressor. |
| *TargetEncoder* | A transformer that encodes categorical features into target encodings. |
| *TimeSeriesBinaryClassificationPipeline* | Pipeline base class for time series binary classification problems. |
| *TimeSeriesClassificationPipeline* | Pipeline base class for time series classification problems. |
| *TimeSeriesFeaturizer* | Transformer that delays input features and target variable for time series problems. |
| *TimeSeriesImputer* | Imputes missing data according to a specified timeseries-specific imputation strategy. |
| *TimeSeriesMulticlassClassificationPipeline* | Pipeline base class for time series multiclass classification problems. |
| *TimeSeriesRegressionPipeline* | Pipeline base class for time series regression problems. |
| *TimeSeriesRegularizer* | Transformer that regularizes an inconsistently spaced datetime column. |
| *Transformer* | A component that may or may not need fitting that transforms data. These components are used before an estimator. |
| *VowpalWabbitBinaryClassifier* | Vowpal Wabbit Binary Classifier. |
| *VowpalWabbitMulticlassClassifier* | Vowpal Wabbit Multiclass Classifier. |
| *VowpalWabbitRegressor* | Vowpal Wabbit Regressor. |
| *XGBoostClassifier* | XGBoost Classifier. |

continues on next page

Table 11 – continued from previous page

| | |
|---|---|
| *XGBoostRegressor* | XGBoost Regressor. |

## Contents

**class** evalml.pipelines.**ARIMARegressor**(*time_index: Optional[Hashable] = None, trend: Optional[str] = None, start_p: int = 2, d: int = 0, start_q: int = 2, max_p: int = 5, max_d: int = 2, max_q: int = 5, seasonal: bool = True, sp: int = 1, n_jobs: int = - 1, random_seed: Union[int, float] = 0, maxiter: int = 10, use_covariates: bool = True, **kwargs*)

Autoregressive Integrated Moving Average Model. The three parameters (p, d, q) are the AR order, the degree of differencing, and the MA order. More information here: https://www.statsmodels.org/devel/generated/statsmodels.tsa.arima.model.ARIMA.html.

Currently ARIMARegressor isn't supported via conda install. It's recommended that it be installed via PyPI.

**Parameters**

- **time_index** (*str*) – Specifies the name of the column in X that provides the datetime objects. Defaults to None.
- **trend** (*str*) – Controls the deterministic trend. Options are ['n', 'c', 't', 'ct'] where 'c' is a constant term, 't' indicates a linear trend, and 'ct' is both. Can also be an iterable when defining a polynomial, such as [1, 1, 0, 1].
- **start_p** (*int*) – Minimum Autoregressive order. Defaults to 2.
- **d** (*int*) – Minimum Differencing degree. Defaults to 0.
- **start_q** (*int*) – Minimum Moving Average order. Defaults to 2.
- **max_p** (*int*) – Maximum Autoregressive order. Defaults to 5.
- **max_d** (*int*) – Maximum Differencing degree. Defaults to 2.
- **max_q** (*int*) – Maximum Moving Average order. Defaults to 5.
- **seasonal** (*boolean*) – Whether to fit a seasonal model to ARIMA. Defaults to True.
- **sp** (*int or str*) – Period for seasonal differencing, specifically the number of periods in each season. If "detect", this model will automatically detect this parameter (given the time series is a standard frequency) and will fall back to 1 (no seasonality) if it cannot be detected. Defaults to 1.
- **n_jobs** (*int or None*) – Non-negative integer describing level of parallelism used for pipelines. Defaults to -1.
- **random_seed** (*int*) – Seed for the random number generator. Defaults to 0.

**Attributes**

| | |
|---|---|
| **hyper-parameter_ranges** | { "start_p": Integer(1, 3), "d": Integer(0, 2), "start_q": Integer(1, 3), "max_p": Integer(3, 10), "max_d": Integer(2, 5), "max_q": Integer(3, 10), "seasonal": [True, False],} |
| **max_cols** | 7 |
| **max_rows** | 1000 |
| **model_family** | ModelFamily.ARIMA |
| **modifies_features** | True |
| **modifies_target** | False |
| **name** | ARIMA Regressor |
| **supported_problem_types** | [ProblemTypes.TIME_SERIES_REGRESSION] |
| **training_only** | False |

**Methods**

| | |
|---|---|
| *clone* | Constructs a new component with the same parameters and random state. |
| *default_parameters* | Returns the default parameters for this component. |
| *describe* | Describe a component and its parameters. |
| *feature_importance* | Returns array of 0's with a length of 1 as feature_importance is not defined for ARIMA regressor. |
| *fit* | Fits ARIMA regressor to data. |
| *get_prediction_intervals* | Find the prediction intervals using the fitted ARIMARegressor. |
| *load* | Loads component at file path. |
| *needs_fitting* | Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances. |
| *parameters* | Returns the parameters which were used to initialize the component. |
| *predict* | Make predictions using fitted ARIMA regressor. |
| *predict_proba* | Make probability estimates for labels. |
| *save* | Saves component at file path. |
| *update_parameters* | Updates the parameter dictionary of the component. |

**clone**(*self*)

Constructs a new component with the same parameters and random state.

**Returns** A new instance of this component with identical parameters and random state.

**default_parameters**(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

**Returns** Default parameters for this component.

**Return type** dict

**describe**(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

**Parameters**

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description as dictionary in the format {"name": name, "parameters": parameters}

**Returns** Returns dictionary if return_dict is True, else None.

**Return type** None or dict

**property feature_importance**(*self*) → numpy.ndarray

Returns array of 0's with a length of 1 as feature_importance is not defined for ARIMA regressor.

**fit**(*self, X: pandas.DataFrame, y: Optional[pandas.Series] = None*)

Fits ARIMA regressor to data.

**Parameters**

- **X** (*pd.DataFrame*) – The input training data of shape [n_samples, n_features].
- **y** (*pd.Series*) – The target training data of length [n_samples].

**Returns** self

**Raises** **ValueError** – If y was not passed in.

**get_prediction_intervals**(*self, X: pandas.DataFrame, y: pandas.Series = None, coverage: List[float] = None, predictions: pandas.Series = None*) → Dict[str, pandas.Series]

Find the prediction intervals using the fitted ARIMARegressor.

**Parameters**

- **X** (*pd.DataFrame*) – Data of shape [n_samples, n_features].
- **y** (*pd.Series*) – Target data. Optional.
- **coverage** (*list[float]*) – A list of floats between the values 0 and 1 that the upper and lower bounds of the prediction interval should be calculated for.
- **predictions** (*pd.Series*) – Not used for ARIMA regressor.

**Returns** Prediction intervals, keys are in the format {coverage}_lower or {coverage}_upper.

**Return type** dict

**static load**(*file_path*)

Loads component at file path.

**Parameters** **file_path** (*str*) – Location to load file.

**Returns** ComponentBase object

**needs_fitting**(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need to be fit or whose fit methods do nothing.

**Returns** True.

**property parameters**(*self*)

Returns the parameters which were used to initialize the component.

**predict**(*self*, *X: pandas.DataFrame*, *y: Optional[pandas.Series] = None*) → pandas.Series

Make predictions using fitted ARIMA regressor.

**Parameters**

- **X** (*pd.DataFrame*) – Data of shape [n_samples, n_features].
- **y** (*pd.Series*) – Target data.

**Returns** Predicted values.

**Return type** pd.Series

**Raises** **ValueError** – If X was passed to *fit* but not passed in *predict*.

**predict_proba**(*self*, *X: pandas.DataFrame*) → pandas.Series

Make probability estimates for labels.

**Parameters** **X** (*pd.DataFrame*) – Features.

**Returns** Probability estimates.

**Return type** pd.Series

**Raises** **MethodPropertyNotFoundError** – If estimator does not have a predict_proba method or a component_obj that implements predict_proba.

**save**(*self*, *file_path*, *pickle_protocol=cloudpickle.DEFAULT_PROTOCOL*)

Saves component at file path.

**Parameters**

- **file_path** (*str*) – Location to save file.
- **pickle_protocol** (*int*) – The pickle data stream format.

**update_parameters**(*self*, *update_dict*, *reset_fit=True*)

Updates the parameter dictionary of the component.

**Parameters**

- **update_dict** (*dict*) – A dict of parameters to update.
- **reset_fit** (*bool, optional*) – If True, will set *_is_fitted* to False.

**class** evalml.pipelines.**BinaryClassificationPipeline**(*component_graph*, *parameters=None*, *custom_name=None*, *random_seed=0*)

Pipeline subclass for all binary classification pipelines.

**Parameters**

- **component_graph** (ComponentGraph, *list, dict*) – ComponentGraph instance, list of components in order, or dictionary of components. Accepts strings or ComponentBase subclasses in the list. Note that when duplicate components are specified in a list, the duplicate component names will be modified with the component's index in the list. For example, the component graph [Imputer, One Hot Encoder, Imputer, Logistic Regression Classifier] will have names ["Imputer", "One Hot Encoder", "Imputer_2", "Logistic Regression Classifier"]
- **parameters** (*dict*) – Dictionary with component names as keys and dictionary of that component's parameters as values. An empty dictionary or None implies using all default values for component parameters. Defaults to None.
- **custom_name** (*str*) – Custom name for the pipeline. Defaults to None.
- **random_seed** (*int*) – Seed for the random number generator. Defaults to 0.

### Example

```
>>> pipeline = BinaryClassificationPipeline(component_graph=["Simple Imputer",
↪"Logistic Regression Classifier"],
...                                          parameters={"Logistic Regression␣
↪Classifier": {"penalty": "elasticnet",
...                                                                            ␣
↪       "solver": "liblinear"}},
...                                          custom_name="My Binary Pipeline")
...
>>> assert pipeline.custom_name == "My Binary Pipeline"
>>> assert pipeline.component_graph.component_dict.keys() == {'Simple Imputer',
↪'Logistic Regression Classifier'}
```

The pipeline parameters will be chosen from the default parameters for every component, unless specific parameters were passed in as they were above.

```
>>> assert pipeline.parameters == {
...     'Simple Imputer': {'impute_strategy': 'most_frequent', 'fill_value': None},
...     'Logistic Regression Classifier': {'penalty': 'elasticnet',
...                                        'C': 1.0,
...                                        'n_jobs': -1,
...                                        'multi_class': 'auto',
...                                        'solver': 'liblinear'}}
```

#### Attributes

| | |
|---|---|
| **problem_type** | ProblemTypes.BINARY |

#### Methods

| | |
|---|---|
| *can_tune_threshold_with_objective* | Determine whether the threshold of a binary classification pipeline can be tuned. |
| *classes_* | Gets the class names for the pipeline. Will return None before pipeline is fit. |
| *clone* | Constructs a new pipeline with the same components, parameters, and random seed. |
| *create_objectives* | Create objective instances from a list of strings or objective classes. |
| *custom_name* | Custom name of the pipeline. |
| *describe* | Outputs pipeline details including component parameters. |
| *feature_importance* | Importance associated with each feature. Features dropped by the feature selection are excluded. |
| *fit* | Build a classification model. For string and categorical targets, classes are sorted by sorted(set(y)) and then are mapped to values between 0 and n_classes-1. |
| *fit_transform* | Fit and transform all components in the component graph, if all components are Transformers. |
| *get_component* | Returns component by name. |
| *get_hyperparameter_ranges* | Returns hyperparameter ranges from all components as a dictionary. |
| *graph* | Generate an image representing the pipeline graph. |
| *graph_dict* | Generates a dictionary with nodes consisting of the component names and parameters, and edges detailing component relationships. This dictionary is JSON serializable in most cases. |
| *graph_feature_importance* | Generate a bar graph of the pipeline's feature importance. |
| *inverse_transform* | Apply component inverse_transform methods to estimator predictions in reverse order. |
| *load* | Loads pipeline at file path. |
| *model_family* | Returns model family of this pipeline. |
| *name* | Name of the pipeline. |
| *new* | Constructs a new instance of the pipeline with the same component graph but with a different set of parameters. Not to be confused with python's __new__ method. |
| *optimize_threshold* | Optimize the pipeline threshold given the objective to use. Only used for binary problems with objectives whose thresholds can be tuned. |
| *parameters* | Parameter dictionary for this pipeline. |
| *predict* | Make predictions using selected features. |
| *predict_proba* | Make probability estimates for labels. Assumes that the column at index 1 represents the positive label case. |
| *save* | Saves pipeline at file path. |
| *score* | Evaluate model performance on objectives. |
| *summary* | A short summary of the pipeline structure, describing the list of components used. |
| *threshold* | Threshold used to make a prediction. Defaults to None. |
| *transform* | Transform the input. |
| *transform_all_but_final* | Transforms the data by applying all pre-processing components. |

**can_tune_threshold_with_objective**(*self*, *objective*)

Determine whether the threshold of a binary classification pipeline can be tuned.

**Parameters** **objective** (*ObjectiveBase*) – Primary AutoMLSearch objective.

**Returns** True if the pipeline threshold can be tuned.

**Return type** bool

**property classes_**(*self*)

Gets the class names for the pipeline. Will return None before pipeline is fit.

**clone**(*self*)

Constructs a new pipeline with the same components, parameters, and random seed.

**Returns** A new instance of this pipeline with identical components, parameters, and random seed.

**static create_objectives**(*objectives*)

Create objective instances from a list of strings or objective classes.

**property custom_name**(*self*)

Custom name of the pipeline.

**describe**(*self*, *return_dict=False*)

Outputs pipeline details including component parameters.

**Parameters** **return_dict** (*bool*) – If True, return dictionary of information about pipeline. Defaults to False.

**Returns** Dictionary of all component parameters if return_dict is True, else None.

**Return type** dict

**property feature_importance**(*self*)

Importance associated with each feature. Features dropped by the feature selection are excluded.

**Returns** Feature names and their corresponding importance

**Return type** pd.DataFrame

**fit**(*self*, *X*, *y*)

Build a classification model. For string and categorical targets, classes are sorted by sorted(set(y)) and then are mapped to values between 0 and n_classes-1.

**Parameters**

- **X** (*pd.DataFrame or np.ndarray*) – The input training data of shape [n_samples, n_features]
- **y** (*pd.Series, np.ndarray*) – The target training labels of length [n_samples]

**Returns** self

**Raises**

- **ValueError** – If the number of unique classes in y are not appropriate for the type of pipeline.
- **TypeError** – If the dtype is boolean but pd.NA exists in the series.
- **Exception** – For all other exceptions.

**fit_transform**(*self*, *X*, *y*)

Fit and transform all components in the component graph, if all components are Transformers.

**Parameters**

- **X** (*pd.DataFrame*) – Input features of shape [n_samples, n_features].
- **y** (*pd.Series*) – The target data of length [n_samples].

**Returns** Transformed output.

**Return type** pd.DataFrame

**Raises** **ValueError** – If final component is an Estimator.

**get_component**(*self*, *name*)

Returns component by name.

**Parameters** **name** (*str*) – Name of component.

**Returns** Component to return

**Return type** Component

**get_hyperparameter_ranges**(*self*, *custom_hyperparameters*)

Returns hyperparameter ranges from all components as a dictionary.

**Parameters** **custom_hyperparameters** (*dict*) – Custom hyperparameters for the pipeline.

**Returns** Dictionary of hyperparameter ranges for each component in the pipeline.

**Return type** dict

**graph**(*self*, *filepath=None*)

Generate an image representing the pipeline graph.

**Parameters** **filepath** (*str, optional*) – Path to where the graph should be saved. If set to None (as by default), the graph will not be saved.

**Returns** Graph object that can be directly displayed in Jupyter notebooks.

**Return type** graphviz.Digraph

**Raises**

- **RuntimeError** – If graphviz is not installed.
- **ValueError** – If path is not writeable.

**graph_dict**(*self*)

Generates a dictionary with nodes consisting of the component names and parameters, and edges detailing component relationships. This dictionary is JSON serializable in most cases.

x_edges specifies from which component feature data is being passed. y_edges specifies from which component target data is being passed. This can be used to build graphs across a variety of visualization tools. Template: {"Nodes": {"component_name": {"Name": class_name, "Parameters": parameters_attributes}, ...}}, "x_edges": [[from_component_name, to_component_name], [from_component_name, to_component_name], ...], "y_edges": [[from_component_name, to_component_name], [from_component_name, to_component_name], ...]}

**Returns** A dictionary representing the DAG structure.

**Return type** dag_dict (dict)

**graph_feature_importance**(*self*, *importance_threshold=0*)

Generate a bar graph of the pipeline's feature importance.

**Parameters** **importance_threshold** (*float, optional*) – If provided, graph features with a permutation importance whose absolute value is larger than importance_threshold. Defaults to zero.

**Returns** A bar graph showing features and their corresponding importance.

**Return type** plotly.Figure

**Raises** **ValueError** – If importance threshold is not valid.

**inverse_transform**(*self*, *y*)

Apply component inverse_transform methods to estimator predictions in reverse order.

Components that implement inverse_transform are PolynomialDecomposer, LogTransformer, LabelEncoder (tbd).

**Parameters** **y** (*pd.Series*) – Final component features.

**Returns** The inverse transform of the target.

**Return type** pd.Series

**static load**(*file_path*)

Loads pipeline at file path.

**Parameters** **file_path** (*str*) – Location to load file.

**Returns** PipelineBase object

**property model_family**(*self*)

Returns model family of this pipeline.

**property name**(*self*)

Name of the pipeline.

**new**(*self*, *parameters*, *random_seed=0*)

Constructs a new instance of the pipeline with the same component graph but with a different set of parameters. Not to be confused with python's __new__ method.

**Parameters**

- **parameters** (*dict*) – Dictionary with component names as keys and dictionary of that component's parameters as values. An empty dictionary or None implies using all default values for component parameters. Defaults to None.
- **random_seed** (*int*) – Seed for the random number generator. Defaults to 0.

**Returns** A new instance of this pipeline with identical components.

**optimize_threshold**(*self*, *X*, *y*, *y_pred_proba*, *objective*)

Optimize the pipeline threshold given the objective to use. Only used for binary problems with objectives whose thresholds can be tuned.

**Parameters**

- **X** (*pd.DataFrame*) – Input features.
- **y** (*pd.Series*) – Input target values.
- **y_pred_proba** (*pd.Series*) – The predicted probabilities of the target outputted by the pipeline.

- **objective** (*ObjectiveBase*) – The objective to threshold with. Must have a tunable threshold.

**Raises** **ValueError** – If objective is not optimizable.

**property parameters**(*self*)

Parameter dictionary for this pipeline.

**Returns** Dictionary of all component parameters.

**Return type** dict

**predict**(*self*, *X*, *objective=None*, *X_train=None*, *y_train=None*)

Make predictions using selected features.

Note: we cast y as ints first to address boolean values that may be returned from calculating predictions which we would not be able to otherwise transform if we originally had integer targets.

**Parameters**

- **X** (*pd.DataFrame*) – Data of shape [n_samples, n_features].
- **objective** (*Object or string*) – The objective to use to make predictions.
- **X_train** (*pd.DataFrame*) – Training data. Ignored. Only used for time series.
- **y_train** (*pd.Series*) – Training labels. Ignored. Only used for time series.

**Returns** Estimated labels.

**Return type** pd.Series

**predict_proba**(*self*, *X*, *X_train=None*, *y_train=None*)

Make probability estimates for labels. Assumes that the column at index 1 represents the positive label case.

**Parameters**

- **X** (*pd.DataFrame or np.ndarray*) – Data of shape [n_samples, n_features]
- **X_train** (*pd.DataFrame or np.ndarray or None*) – Training data. Ignored. Only used for time series.
- **y_train** (*pd.Series or None*) – Training labels. Ignored. Only used for time series.

**Returns** Probability estimates

**Return type** pd.Series

**save**(*self*, *file_path*, *pickle_protocol=cloudpickle.DEFAULT_PROTOCOL*)

Saves pipeline at file path.

**Parameters**

- **file_path** (*str*) – Location to save file.
- **pickle_protocol** (*int*) – The pickle data stream format.

**score**(*self*, *X*, *y*, *objectives*, *X_train=None*, *y_train=None*)

Evaluate model performance on objectives.

**Parameters**

- **X** (*pd.DataFrame*) – Data of shape [n_samples, n_features]
- **y** (*pd.Series*) – True labels of length [n_samples]

- **objectives** (*list*) – List of objectives to score
- **X_train** (*pd.DataFrame*) – Training data. Ignored. Only used for time series.
- **y_train** (*pd.Series*) – Training labels. Ignored. Only used for time series.

**Returns** Ordered dictionary of objective scores.

**Return type** dict

**property summary**(*self*)

A short summary of the pipeline structure, describing the list of components used.

Example: Logistic Regression Classifier w/ Simple Imputer + One Hot Encoder

**Returns** A string describing the pipeline structure.

**property threshold**(*self*)

Threshold used to make a prediction. Defaults to None.

**transform**(*self*, *X*, *y=None*)

Transform the input.

**Parameters**

- **X** (*pd.DataFrame, or np.ndarray*) – Data of shape [n_samples, n_features].
- **y** (*pd.Series*) – The target data of length [n_samples]. Defaults to None.

**Returns** Transformed output.

**Return type** pd.DataFrame

**transform_all_but_final**(*self*, *X*, *y=None*, *X_train=None*, *y_train=None*)

Transforms the data by applying all pre-processing components.

**Parameters**

- **X** (*pd.DataFrame*) – Input data to the pipeline to transform.
- **y** (*pd.Series or None*) – Targets corresponding to X. Optional.
- **X_train** (*pd.DataFrame or np.ndarray or None*) – Training data. Only used for time series.
- **y_train** (*pd.Series or None*) – Training labels. Only used for time series.

**Returns** New transformed features.

**Return type** pd.DataFrame

**class** evalml.pipelines.**CatBoostClassifier**(*n_estimators=10*, *eta=0.03*, *max_depth=6*, *bootstrap_type=None*, *silent=True*, *allow_writing_files=False*, *random_seed=0*, *n_jobs=- 1*, ***kwargs*)

CatBoost Classifier, a classifier that uses gradient-boosting on decision trees. CatBoost is an open-source library and natively supports categorical features.

For more information, check out https://catboost.ai/

**Parameters**

- **n_estimators** (*float*) – The maximum number of trees to build. Defaults to 10.
- **eta** (*float*) – The learning rate. Defaults to 0.03.
- **max_depth** (*int*) – The maximum tree depth for base learners. Defaults to 6.

- **bootstrap_type** (*string*) – Defines the method for sampling the weights of objects. Available methods are 'Bayesian', 'Bernoulli', 'MVS'. Defaults to None.
- **silent** (*boolean*) – Whether to use the "silent" logging mode. Defaults to True.
- **allow_writing_files** (*boolean*) – Whether to allow writing snapshot files while training. Defaults to False.
- **n_jobs** (*int or None*) – Number of jobs to run in parallel. -1 uses all processes. Defaults to -1.
- **random_seed** (*int*) – Seed for the random number generator. Defaults to 0.

**Attributes**

| | |
|---|---|
| **hyper-parameter_ranges** | { "n_estimators": Integer(4, 100), "eta": Real(0.000001, 1), "max_depth": Integer(4, 10),} |
| **model_family** | ModelFamily.CATBOOST |
| **modifies_features** | True |
| **modifies_target** | False |
| **name** | CatBoost Classifier |
| **supported_problem_types** | [ ProblemTypes.BINARY, ProblemTypes.MULTICLASS, ProblemTypes.TIME_SERIES_BINARY, ProblemTypes.TIME_SERIES_MULTICLASS,] |
| **training_only** | False |

**Methods**

| | |
|---|---|
| *clone* | Constructs a new component with the same parameters and random state. |
| *default_parameters* | Returns the default parameters for this component. |
| *describe* | Describe a component and its parameters. |
| *feature_importance* | Feature importance of fitted CatBoost classifier. |
| *fit* | Fits CatBoost classifier component to data. |
| *get_prediction_intervals* | Find the prediction intervals using the fitted regressor. |
| *load* | Loads component at file path. |
| *needs_fitting* | Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances. |
| *parameters* | Returns the parameters which were used to initialize the component. |
| *predict* | Make predictions using the fitted CatBoost classifier. |
| *predict_proba* | Make prediction probabilities using the fitted CatBoost classifier. |
| *save* | Saves component at file path. |
| *update_parameters* | Updates the parameter dictionary of the component. |

**clone**(*self*)

Constructs a new component with the same parameters and random state.

**Returns** A new instance of this component with identical parameters and random state.

**default_parameters**(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

**Returns** Default parameters for this component.

**Return type** dict

**describe**(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

**Parameters**

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description as dictionary in the format {"name": name, "parameters": parameters}

**Returns** Returns dictionary if return_dict is True, else None.

**Return type** None or dict

**property feature_importance**(*self*)

Feature importance of fitted CatBoost classifier.

**fit**(*self*, *X*, *y=None*)

Fits CatBoost classifier component to data.

**Parameters**

- **X** (*pd.DataFrame*) – The input training data of shape [n_samples, n_features].
- **y** (*pd.Series*) – The target training data of length [n_samples].

**Returns** self

**get_prediction_intervals**(*self*, *X: pandas.DataFrame*, *y: Optional[pandas.Series] = None*, *coverage: List[float] = None*, *predictions: pandas.Series = None*) → Dict[str, pandas.Series]

Find the prediction intervals using the fitted regressor.

This function takes the predictions of the fitted estimator and calculates the rolling standard deviation across all predictions using a window size of 5. The lower and upper predictions are determined by taking the percent point (quantile) function of the lower tail probability at each bound multiplied by the rolling standard deviation.

**Parameters**

- **X** (*pd.DataFrame*) – Data of shape [n_samples, n_features].
- **y** (*pd.Series*) – Target data. Ignored.
- **coverage** (*list[float]*) – A list of floats between the values 0 and 1 that the upper and lower bounds of the prediction interval should be calculated for.
- **predictions** (*pd.Series*) – Optional list of predictions to use. If None, will generate predictions using *X*.

**Returns** Prediction intervals, keys are in the format {coverage}_lower or {coverage}_upper.

**Return type** dict

**Raises** **MethodPropertyNotFoundError** – If the estimator does not support Time Series Regression as a problem type.

**static load**(*file_path*)

Loads component at file path.

**Parameters** **file_path** (*str*) – Location to load file.

**Returns** ComponentBase object

**needs_fitting**(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need to be fit or whose fit methods do nothing.

**Returns** True.

**property parameters**(*self*)

Returns the parameters which were used to initialize the component.

**predict**(*self*, *X*)

Make predictions using the fitted CatBoost classifier.

**Parameters** **X** (*pd.DataFrame*) – Data of shape [n_samples, n_features].

**Returns** Predicted values.

**Return type** pd.Series

**predict_proba**(*self*, *X*)

Make prediction probabilities using the fitted CatBoost classifier.

**Parameters** **X** (*pd.DataFrame*) – Data of shape [n_samples, n_features].

**Returns** Predicted probability values.

**Return type** pd.DataFrame

**save**(*self*, *file_path*, *pickle_protocol=cloudpickle.DEFAULT_PROTOCOL*)

Saves component at file path.

**Parameters**

- **file_path** (*str*) – Location to save file.
- **pickle_protocol** (*int*) – The pickle data stream format.

**update_parameters**(*self*, *update_dict*, *reset_fit=True*)

Updates the parameter dictionary of the component.

**Parameters**

- **update_dict** (*dict*) – A dict of parameters to update.
- **reset_fit** (*bool, optional*) – If True, will set *_is_fitted* to False.

**class** evalml.pipelines.**CatBoostRegressor**(*n_estimators=10*, *eta=0.03*, *max_depth=6*, *bootstrap_type=None*, *silent=False*, *allow_writing_files=False*, *random_seed=0*, *n_jobs=- 1*, ***kwargs*)

CatBoost Regressor, a regressor that uses gradient-boosting on decision trees. CatBoost is an open-source library and natively supports categorical features.

For more information, check out https://catboost.ai/

**Parameters**

- **n_estimators** (*float*) – The maximum number of trees to build. Defaults to 10.

- **eta** (*float*) – The learning rate. Defaults to 0.03.
- **max_depth** (*int*) – The maximum tree depth for base learners. Defaults to 6.
- **bootstrap_type** (*string*) – Defines the method for sampling the weights of objects. Available methods are 'Bayesian', 'Bernoulli', 'MVS'. Defaults to None.
- **silent** (*boolean*) – Whether to use the "silent" logging mode. Defaults to True.
- **allow_writing_files** (*boolean*) – Whether to allow writing snapshot files while training. Defaults to False.
- **n_jobs** (*int or None*) – Number of jobs to run in parallel. -1 uses all processes. Defaults to -1.
- **random_seed** (*int*) – Seed for the random number generator. Defaults to 0.

**Attributes**

| | |
|---|---|
| **hyper-parameter_ranges** | { "n_estimators": Integer(4, 100), "eta": Real(0.000001, 1), "max_depth": Integer(4, 10),} |
| **model_family** | ModelFamily.CATBOOST |
| **modifies_features** | True |
| **modifies_target** | False |
| **name** | CatBoost Regressor |
| **supported_problem_types** | [ ProblemTypes.REGRESSION, ProblemTypes.TIME_SERIES_REGRESSION,] |
| **training_only** | False |

**Methods**

| | |
|---|---|
| *clone* | Constructs a new component with the same parameters and random state. |
| *default_parameters* | Returns the default parameters for this component. |
| *describe* | Describe a component and its parameters. |
| *feature_importance* | Feature importance of fitted CatBoost regressor. |
| *fit* | Fits CatBoost regressor component to data. |
| *get_prediction_intervals* | Find the prediction intervals using the fitted regressor. |
| *load* | Loads component at file path. |
| *needs_fitting* | Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances. |
| *parameters* | Returns the parameters which were used to initialize the component. |
| *predict* | Make predictions using the fitted CatBoost regressor. |
| *predict_proba* | Make probability estimates for labels. |
| *save* | Saves component at file path. |
| *update_parameters* | Updates the parameter dictionary of the component. |

**clone**(*self*)

Constructs a new component with the same parameters and random state.

**Returns** A new instance of this component with identical parameters and random state.

**default_parameters**(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

**Returns** Default parameters for this component.

**Return type** dict

**describe**(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

**Parameters**

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description as dictionary in the format {"name": name, "parameters": parameters}

**Returns** Returns dictionary if return_dict is True, else None.

**Return type** None or dict

**property feature_importance**(*self*)

Feature importance of fitted CatBoost regressor.

**fit**(*self*, *X*, *y=None*)

Fits CatBoost regressor component to data.

**Parameters**

- **X** (*pd.DataFrame*) – The input training data of shape [n_samples, n_features].
- **y** (*pd.Series*) – The target training data of length [n_samples].

**Returns** self

**get_prediction_intervals**(*self*, *X: pandas.DataFrame*, *y: Optional[pandas.Series] = None*, *coverage: List[float] = None*, *predictions: pandas.Series = None*) → Dict[str, pandas.Series]

Find the prediction intervals using the fitted regressor.

This function takes the predictions of the fitted estimator and calculates the rolling standard deviation across all predictions using a window size of 5. The lower and upper predictions are determined by taking the percent point (quantile) function of the lower tail probability at each bound multiplied by the rolling standard deviation.

**Parameters**

- **X** (*pd.DataFrame*) – Data of shape [n_samples, n_features].
- **y** (*pd.Series*) – Target data. Ignored.
- **coverage** (*list[float]*) – A list of floats between the values 0 and 1 that the upper and lower bounds of the prediction interval should be calculated for.
- **predictions** (*pd.Series*) – Optional list of predictions to use. If None, will generate predictions using *X*.

**Returns** Prediction intervals, keys are in the format {coverage}_lower or {coverage}_upper.

**Return type** dict

**Raises** `MethodPropertyNotFoundError` – If the estimator does not support Time Series Regression as a problem type.

`static load`(*file_path*)

Loads component at file path.

**Parameters** `file_path` (*str*) – Location to load file.

**Returns** ComponentBase object

`needs_fitting`(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need to be fit or whose fit methods do nothing.

**Returns** True.

`property parameters`(*self*)

Returns the parameters which were used to initialize the component.

`predict`(*self*, *X*)

Make predictions using the fitted CatBoost regressor.

**Parameters** `X` (*pd.DataFrame*) – Data of shape [n_samples, n_features].

**Returns** Predicted values.

**Return type** pd.DataFrame

`predict_proba`(*self*, *X: pandas.DataFrame*) → pandas.Series

Make probability estimates for labels.

**Parameters** `X` (*pd.DataFrame*) – Features.

**Returns** Probability estimates.

**Return type** pd.Series

**Raises** `MethodPropertyNotFoundError` – If estimator does not have a predict_proba method or a component_obj that implements predict_proba.

`save`(*self*, *file_path*, *pickle_protocol=cloudpickle.DEFAULT_PROTOCOL*)

Saves component at file path.

**Parameters**

- `file_path` (*str*) – Location to save file.
- `pickle_protocol` (*int*) – The pickle data stream format.

`update_parameters`(*self*, *update_dict*, *reset_fit=True*)

Updates the parameter dictionary of the component.

**Parameters**

- `update_dict` (*dict*) – A dict of parameters to update.
- `reset_fit` (*bool, optional*) – If True, will set *_is_fitted* to False.

`class evalml.pipelines.ClassificationPipeline`(*component_graph*, *parameters=None*, *custom_name=None*, *random_seed=0*)

Pipeline subclass for all classification pipelines.

**Parameters**

- **component_graph** (*list or dict*) – List of components in order. Accepts strings or ComponentBase subclasses in the list. Note that when duplicate components are specified in a list, the duplicate component names will be modified with the component's index in the list. For example, the component graph [Imputer, One Hot Encoder, Imputer, Logistic Regression Classifier] will have names ["Imputer", "One Hot Encoder", "Imputer_2", "Logistic Regression Classifier"]
- **parameters** (*dict*) – Dictionary with component names as keys and dictionary of that component's parameters as values. An empty dictionary or None implies using all default values for component parameters. Defaults to None.
- **custom_name** (*str*) – Custom name for the pipeline. Defaults to None.
- **random_seed** (*int*) – Seed for the random number generator. Defaults to 0.

**Attributes**

| | |
|---|---|
| **problem_type** | None |

**Methods**

| | |
|---|---|
| *can_tune_threshold_with_objective* | Determine whether the threshold of a binary classification pipeline can be tuned. |
| *classes_* | Gets the class names for the pipeline. Will return None before pipeline is fit. |
| *clone* | Constructs a new pipeline with the same components, parameters, and random seed. |
| *create_objectives* | Create objective instances from a list of strings or objective classes. |
| *custom_name* | Custom name of the pipeline. |
| *describe* | Outputs pipeline details including component parameters. |
| *feature_importance* | Importance associated with each feature. Features dropped by the feature selection are excluded. |
| *fit* | Build a classification model. For string and categorical targets, classes are sorted by sorted(set(y)) and then are mapped to values between 0 and n_classes-1. |
| *fit_transform* | Fit and transform all components in the component graph, if all components are Transformers. |
| *get_component* | Returns component by name. |
| *get_hyperparameter_ranges* | Returns hyperparameter ranges from all components as a dictionary. |
| *graph* | Generate an image representing the pipeline graph. |
| *graph_dict* | Generates a dictionary with nodes consisting of the component names and parameters, and edges detailing component relationships. This dictionary is JSON serializable in most cases. |
| *graph_feature_importance* | Generate a bar graph of the pipeline's feature importance. |
| *inverse_transform* | Apply component inverse_transform methods to estimator predictions in reverse order. |
| *load* | Loads pipeline at file path. |
| *model_family* | Returns model family of this pipeline. |
| *name* | Name of the pipeline. |
| *new* | Constructs a new instance of the pipeline with the same component graph but with a different set of parameters. Not to be confused with python's __new__ method. |
| *parameters* | Parameter dictionary for this pipeline. |
| *predict* | Make predictions using selected features. |
| *predict_proba* | Make probability estimates for labels. |
| *save* | Saves pipeline at file path. |
| *score* | Evaluate model performance on objectives. |
| *summary* | A short summary of the pipeline structure, describing the list of components used. |
| *transform* | Transform the input. |
| *transform_all_but_final* | Transforms the data by applying all pre-processing components. |

**can_tune_threshold_with_objective**(*self*, *objective*)

Determine whether the threshold of a binary classification pipeline can be tuned.

**Parameters** **objective** (*ObjectiveBase*) – Primary AutoMLSearch objective.

**Returns** True if the pipeline threshold can be tuned.

**Return type** bool

**property classes_**(*self*)

Gets the class names for the pipeline. Will return None before pipeline is fit.

**clone**(*self*)

Constructs a new pipeline with the same components, parameters, and random seed.

**Returns** A new instance of this pipeline with identical components, parameters, and random seed.

**static create_objectives**(*objectives*)

Create objective instances from a list of strings or objective classes.

**property custom_name**(*self*)

Custom name of the pipeline.

**describe**(*self*, *return_dict=False*)

Outputs pipeline details including component parameters.

**Parameters** **return_dict** (*bool*) – If True, return dictionary of information about pipeline. Defaults to False.

**Returns** Dictionary of all component parameters if return_dict is True, else None.

**Return type** dict

**property feature_importance**(*self*)

Importance associated with each feature. Features dropped by the feature selection are excluded.

**Returns** Feature names and their corresponding importance

**Return type** pd.DataFrame

**fit**(*self*, *X*, *y*)

Build a classification model. For string and categorical targets, classes are sorted by sorted(set(y)) and then are mapped to values between 0 and n_classes-1.

**Parameters**

- **X** (*pd.DataFrame or np.ndarray*) – The input training data of shape [n_samples, n_features]
- **y** (*pd.Series, np.ndarray*) – The target training labels of length [n_samples]

**Returns** self

**Raises**

- **ValueError** – If the number of unique classes in y are not appropriate for the type of pipeline.
- **TypeError** – If the dtype is boolean but pd.NA exists in the series.
- **Exception** – For all other exceptions.

**fit_transform**(*self*, *X*, *y*)

Fit and transform all components in the component graph, if all components are Transformers.

**Parameters**

- **X** (*pd.DataFrame*) – Input features of shape [n_samples, n_features].
- **y** (*pd.Series*) – The target data of length [n_samples].

**Returns** Transformed output.

**Return type** pd.DataFrame

**Raises** **ValueError** – If final component is an Estimator.

**get_component**(*self*, *name*)

Returns component by name.

**Parameters** **name** (*str*) – Name of component.

**Returns** Component to return

**Return type** Component

**get_hyperparameter_ranges**(*self*, *custom_hyperparameters*)

Returns hyperparameter ranges from all components as a dictionary.

**Parameters** **custom_hyperparameters** (*dict*) – Custom hyperparameters for the pipeline.

**Returns** Dictionary of hyperparameter ranges for each component in the pipeline.

**Return type** dict

**graph**(*self*, *filepath=None*)

Generate an image representing the pipeline graph.

**Parameters** **filepath** (*str, optional*) – Path to where the graph should be saved. If set to None (as by default), the graph will not be saved.

**Returns** Graph object that can be directly displayed in Jupyter notebooks.

**Return type** graphviz.Digraph

**Raises**

- **RuntimeError** – If graphviz is not installed.
- **ValueError** – If path is not writeable.

**graph_dict**(*self*)

Generates a dictionary with nodes consisting of the component names and parameters, and edges detailing component relationships. This dictionary is JSON serializable in most cases.

x_edges specifies from which component feature data is being passed. y_edges specifies from which component target data is being passed. This can be used to build graphs across a variety of visualization tools. Template: {"Nodes": {"component_name": {"Name": class_name, "Parameters": parameters_attributes}, ...}}, "x_edges": [[from_component_name, to_component_name], [from_component_name, to_component_name], ...], "y_edges": [[from_component_name, to_component_name], [from_component_name, to_component_name], ...]}

**Returns** A dictionary representing the DAG structure.

**Return type** dag_dict (dict)

**graph_feature_importance**(*self*, *importance_threshold=0*)

Generate a bar graph of the pipeline's feature importance.

**Parameters** **importance_threshold** (*float, optional*) – If provided, graph features with a permutation importance whose absolute value is larger than importance_threshold. Defaults to zero.

**Returns** A bar graph showing features and their corresponding importance.

**Return type** plotly.Figure

**Raises** `ValueError` – If importance threshold is not valid.

`inverse_transform`(*self*, *y*)

Apply component inverse_transform methods to estimator predictions in reverse order.

Components that implement inverse_transform are PolynomialDecomposer, LogTransformer, LabelEncoder (tbd).

**Parameters** `y` (*pd.Series*) – Final component features.

**Returns** The inverse transform of the target.

**Return type** pd.Series

`static load`(*file_path*)

Loads pipeline at file path.

**Parameters** `file_path` (*str*) – Location to load file.

**Returns** PipelineBase object

`property model_family`(*self*)

Returns model family of this pipeline.

`property name`(*self*)

Name of the pipeline.

`new`(*self*, *parameters*, *random_seed=0*)

Constructs a new instance of the pipeline with the same component graph but with a different set of parameters. Not to be confused with python's __new__ method.

**Parameters**

- `parameters` (*dict*) – Dictionary with component names as keys and dictionary of that component's parameters as values. An empty dictionary or None implies using all default values for component parameters. Defaults to None.
- `random_seed` (*int*) – Seed for the random number generator. Defaults to 0.

**Returns** A new instance of this pipeline with identical components.

`property parameters`(*self*)

Parameter dictionary for this pipeline.

**Returns** Dictionary of all component parameters.

**Return type** dict

`predict`(*self*, *X*, *objective=None*, *X_train=None*, *y_train=None*)

Make predictions using selected features.

Note: we cast y as ints first to address boolean values that may be returned from calculating predictions which we would not be able to otherwise transform if we originally had integer targets.

**Parameters**

- `X` (*pd.DataFrame*) – Data of shape [n_samples, n_features].
- `objective` (*Object or string*) – The objective to use to make predictions.
- `X_train` (*pd.DataFrame*) – Training data. Ignored. Only used for time series.

- **y_train** (*pd.Series*) – Training labels. Ignored. Only used for time series.

**Returns** Estimated labels.

**Return type** pd.Series

**predict_proba**(*self*, *X*, *X_train=None*, *y_train=None*)

Make probability estimates for labels.

**Parameters**

- **X** (*pd.DataFrame or np.ndarray*) – Data of shape [n_samples, n_features]
- **X_train** (*pd.DataFrame or np.ndarray or None*) – Training data. Ignored. Only used for time series.
- **y_train** (*pd.Series or None*) – Training labels. Ignored. Only used for time series.

**Returns** Probability estimates

**Return type** pd.DataFrame

**Raises** **ValueError** – If final component is not an estimator.

**save**(*self*, *file_path*, *pickle_protocol=cloudpickle.DEFAULT_PROTOCOL*)

Saves pipeline at file path.

**Parameters**

- **file_path** (*str*) – Location to save file.
- **pickle_protocol** (*int*) – The pickle data stream format.

**score**(*self*, *X*, *y*, *objectives*, *X_train=None*, *y_train=None*)

Evaluate model performance on objectives.

**Parameters**

- **X** (*pd.DataFrame*) – Data of shape [n_samples, n_features]
- **y** (*pd.Series*) – True labels of length [n_samples]
- **objectives** (*list*) – List of objectives to score
- **X_train** (*pd.DataFrame*) – Training data. Ignored. Only used for time series.
- **y_train** (*pd.Series*) – Training labels. Ignored. Only used for time series.

**Returns** Ordered dictionary of objective scores.

**Return type** dict

**property summary**(*self*)

A short summary of the pipeline structure, describing the list of components used.

Example: Logistic Regression Classifier w/ Simple Imputer + One Hot Encoder

**Returns** A string describing the pipeline structure.

**transform**(*self*, *X*, *y=None*)

Transform the input.

**Parameters**

- **X** (*pd.DataFrame, or np.ndarray*) – Data of shape [n_samples, n_features].
- **y** (*pd.Series*) – The target data of length [n_samples]. Defaults to None.

**Returns** Transformed output.

**Return type** pd.DataFrame

**transform_all_but_final**(*self*, *X*, *y=None*, *X_train=None*, *y_train=None*)

Transforms the data by applying all pre-processing components.

**Parameters**

- **X** (*pd.DataFrame*) – Input data to the pipeline to transform.
- **y** (*pd.Series or None*) – Targets corresponding to X. Optional.
- **X_train** (*pd.DataFrame or np.ndarray or None*) – Training data. Only used for time series.
- **y_train** (*pd.Series or None*) – Training labels. Only used for time series.

**Returns** New transformed features.

**Return type** pd.DataFrame

**class** evalml.pipelines.**ComponentGraph**(*component_dict=None*, *cached_data=None*, *random_seed=0*)

Component graph for a pipeline as a directed acyclic graph (DAG).

**Parameters**

- **component_dict** (*dict*) – A dictionary which specifies the components and edges between components that should be used to create the component graph. Defaults to None.
- **cached_data** (*dict*) – A dictionary of nested cached data. If the hashes and components are in this cache, we skip fitting for these components. Expected to be of format {hash1: {component_name: trained_component, ...}, hash2: {...}, ...}. Defaults to None.
- **random_seed** (*int*) – Seed for the random number generator. Defaults to 0.

### Examples

```
>>> component_dict = {'Imputer': ['Imputer', 'X', 'y'],
...                   'Logistic Regression': ['Logistic Regression Classifier',
↪'Imputer.x', 'y']}
>>> component_graph = ComponentGraph(component_dict)
>>> assert component_graph.compute_order == ['Imputer', 'Logistic Regression']
...
...
>>> component_dict = {'Imputer': ['Imputer', 'X', 'y'],
...                   'OHE': ['One Hot Encoder', 'Imputer.x', 'y'],
...                   'estimator_1': ['Random Forest Classifier', 'OHE.x', 'y'],
...                   'estimator_2': ['Decision Tree Classifier', 'OHE.x', 'y'],
...                   'final': ['Logistic Regression Classifier', 'estimator_1.x',
↪'estimator_2.x', 'y']}
>>> component_graph = ComponentGraph(component_dict)
```

The default parameters for every component in the component graph.

```
>>> assert component_graph.default_parameters == {
...     'Imputer': {'categorical_impute_strategy': 'most_frequent',
...                 'numeric_impute_strategy': 'mean',
...                 'boolean_impute_strategy': 'most_frequent',
```

(continues on next page)

(continued from previous page)

```
...                 'categorical_fill_value': None,
...                 'numeric_fill_value': None,
...                 'boolean_fill_value': None},
...     'One Hot Encoder': {'top_n': 10,
...                         'features_to_encode': None,
...                         'categories': None,
...                         'drop': 'if_binary',
...                         'handle_unknown': 'ignore',
...                         'handle_missing': 'error'},
...     'Random Forest Classifier': {'n_estimators': 100,
...                                  'max_depth': 6,
...                                  'n_jobs': -1},
...     'Decision Tree Classifier': {'criterion': 'gini',
...                                  'max_features': 'auto',
...                                  'max_depth': 6,
...                                  'min_samples_split': 2,
...                                  'min_weight_fraction_leaf': 0.0},
...     'Logistic Regression Classifier': {'penalty': 'l2',
...                                        'C': 1.0,
...                                        'n_jobs': -1,
...                                        'multi_class': 'auto',
...                                        'solver': 'lbfgs'}}
```

**Methods**

| | |
|---|---|
| *compute_order* | The order that components will be computed or called in. |
| *default_parameters* | The default parameter dictionary for this pipeline. |
| *describe* | Outputs component graph details including component parameters. |
| *fit* | Fit each component in the graph. |
| *fit_and_transform_all_but_final* | Fit and transform all components save the final one, usually an estimator. |
| *fit_transform* | Fit and transform all components in the component graph, if all components are Transformers. |
| *generate_order* | Regenerated the topologically sorted order of the graph. |
| *get_component* | Retrieves a single component object from the graph. |
| *get_component_input_logical_types* | Get the logical types that are passed to the given component. |
| *get_estimators* | Gets a list of all the estimator components within this graph. |
| *get_inputs* | Retrieves all inputs for a given component. |
| *get_last_component* | Retrieves the component that is computed last in the graph, usually the final estimator. |
| *graph* | Generate an image representing the component graph. |
| *has_dfs* | Whether this component graph contains a DFSTransformer or not. |
| *instantiate* | Instantiates all uninstantiated components within the graph using the given parameters. An error will be raised if a component is already instantiated but the parameters dict contains arguments for that component. |
| *inverse_transform* | Apply component inverse_transform methods to estimator predictions in reverse order. |
| *last_component_input_logical_types* | Get the logical types that are passed to the last component in the pipeline. |
| *predict* | Make predictions using selected features. |
| *transform* | Transform the input using the component graph. |
| *transform_all_but_final* | Transform all components save the final one, and gathers the data from any number of parents to get all the information that should be fed to the final component. |

**property compute_order**(*self*)

The order that components will be computed or called in.

**property default_parameters**(*self*)

The default parameter dictionary for this pipeline.

**Returns** Dictionary of all component default parameters.

**Return type** dict

**describe**(*self*, *return_dict=False*)

Outputs component graph details including component parameters.

**Parameters** **return_dict** (*bool*) – If True, return dictionary of information about component

graph. Defaults to False.

**Returns** Dictionary of all component parameters if return_dict is True, else None

**Return type** dict

**Raises** `ValueError` – If the componentgraph is not instantiated

`fit`(*self*, *X*, *y*)

Fit each component in the graph.

**Parameters**

- `X` (*pd.DataFrame*) – The input training data of shape [n_samples, n_features].
- `y` (*pd.Series*) – The target training data of length [n_samples].

**Returns** self

`fit_and_transform_all_but_final`(*self*, *X*, *y*)

Fit and transform all components save the final one, usually an estimator.

**Parameters**

- `X` (*pd.DataFrame*) – The input training data of shape [n_samples, n_features].
- `y` (*pd.Series*) – The target training data of length [n_samples].

**Returns** Transformed features and target.

**Return type** Tuple (pd.DataFrame, pd.Series)

`fit_transform`(*self*, *X*, *y*)

Fit and transform all components in the component graph, if all components are Transformers.

**Parameters**

- `X` (*pd.DataFrame*) – Input features of shape [n_samples, n_features].
- `y` (*pd.Series*) – The target data of length [n_samples].

**Returns** Transformed output.

**Return type** pd.DataFrame

**Raises** `ValueError` – If final component is an Estimator.

`classmethod generate_order`(*cls*, *component_dict*)

Regenerated the topologically sorted order of the graph.

`get_component`(*self*, *component_name*)

Retrieves a single component object from the graph.

**Parameters** `component_name` (*str*) – Name of the component to retrieve

**Returns** ComponentBase object

**Raises** `ValueError` – If the component is not in the graph.

`get_component_input_logical_types`(*self*, *component_name*)

Get the logical types that are passed to the given component.

**Parameters** `component_name` (*str*) – Name of component in the graph

**Returns** Dict - Mapping feature name to logical type instance.

**Raises**

- **ValueError** – If the component is not in the graph.
- **ValueError** – If the component graph as not been fitted

**get_estimators**(*self*)

Gets a list of all the estimator components within this graph.

**Returns** All estimator objects within the graph.

**Return type** list

**Raises** **ValueError** – If the component graph is not yet instantiated.

**get_inputs**(*self*, *component_name*)

Retrieves all inputs for a given component.

**Parameters** **component_name** (*str*) – Name of the component to look up.

**Returns** List of inputs for the component to use.

**Return type** list[str]

**Raises** **ValueError** – If the component is not in the graph.

**get_last_component**(*self*)

Retrieves the component that is computed last in the graph, usually the final estimator.

**Returns** ComponentBase object

**Raises** **ValueError** – If the component graph has no edges.

**graph**(*self*, *name=None*, *graph_format=None*)

Generate an image representing the component graph.

**Parameters**

- **name** (*str*) – Name of the graph. Defaults to None.
- **graph_format** (*str*) – file format to save the graph in. Defaults to None.

**Returns** Graph object that can be directly displayed in Jupyter notebooks.

**Return type** graphviz.Digraph

**Raises** **RuntimeError** – If graphviz is not installed.

**property has_dfs**(*self*)

Whether this component graph contains a DFSTransformer or not.

**instantiate**(*self*, *parameters=None*)

Instantiates all uninstantiated components within the graph using the given parameters. An error will be raised if a component is already instantiated but the parameters dict contains arguments for that component.

**Parameters** **parameters** (*dict*) – Dictionary with component names as keys and dictionary of that component's parameters as values. An empty dictionary {} or None implies using all default values for component parameters. If a component in the component graph is already instantiated, it will not use any of its parameters defined in this dictionary. Defaults to None.

**Returns** self

**Raises** **ValueError** – If component graph is already instantiated or if a component errored while instantiating.

**inverse_transform**(*self*, *y*)

Apply component inverse_transform methods to estimator predictions in reverse order.

Components that implement inverse_transform are PolynomialDecomposer, LogTransformer, LabelEncoder (tbd).

**Parameters** **y** – (pd.Series): Final component features.

**Returns** The target with inverse transformation applied.

**Return type** pd.Series

**property last_component_input_logical_types**(*self*)

Get the logical types that are passed to the last component in the pipeline.

**Returns** Dict - Mapping feature name to logical type instance.

**Raises**

- **ValueError** – If the component is not in the graph.
- **ValueError** – If the component graph as not been fitted

**predict**(*self*, *X*)

Make predictions using selected features.

**Parameters** **X** (*pd.DataFrame*) – Input features of shape [n_samples, n_features].

**Returns** Predicted values.

**Return type** pd.Series

**Raises** **ValueError** – If final component is not an Estimator.

**transform**(*self*, *X*, *y=None*)

Transform the input using the component graph.

**Parameters**

- **X** (*pd.DataFrame*) – Input features of shape [n_samples, n_features].
- **y** (*pd.Series*) – The target data of length [n_samples]. Defaults to None.

**Returns** Transformed output.

**Return type** pd.DataFrame

**Raises** **ValueError** – If final component is not a Transformer.

**transform_all_but_final**(*self*, *X*, *y=None*)

Transform all components save the final one, and gathers the data from any number of parents to get all the information that should be fed to the final component.

**Parameters**

- **X** (*pd.DataFrame*) – Data of shape [n_samples, n_features].
- **y** (*pd.Series*) – The target training data of length [n_samples]. Defaults to None.

**Returns** Transformed values.

**Return type** pd.DataFrame

**class** evalml.pipelines.**DecisionTreeClassifier**(*criterion='gini', max_features='auto', max_depth=6, min_samples_split=2, min_weight_fraction_leaf=0.0, random_seed=0, **kwargs*)

Decision Tree Classifier.

**Parameters**

- **criterion** (*{"gini", "entropy"}*) – The function to measure the quality of a split. Supported criteria are "gini" for the Gini impurity and "entropy" for the information gain. Defaults to "gini".
- **max_features** (*int, float or {"auto", "sqrt", "log2"}*) – The number of features to consider when looking for the best split:
  - If int, then consider max_features features at each split.
  - If float, then max_features is a fraction and int(max_features * n_features) features are considered at each split.
  - If "auto", then max_features=sqrt(n_features).
  - If "sqrt", then max_features=sqrt(n_features).
  - If "log2", then max_features=log2(n_features).
  - If None, then max_features = n_features.

  The search for a split does not stop until at least one valid partition of the node samples is found, even if it requires to effectively inspect more than max_features features. Defaults to "auto".
- **max_depth** (*int*) – The maximum depth of the tree. Defaults to 6.
- **min_samples_split** (*int or float*) – The minimum number of samples required to split an internal node:
  - If int, then consider min_samples_split as the minimum number.
  - If float, then min_samples_split is a fraction and ceil(min_samples_split * n_samples) are the minimum number of samples for each split.

  Defaults to 2.
- **min_weight_fraction_leaf** (*float*) – The minimum weighted fraction of the sum total of weights (of all the input samples) required to be at a leaf node. Defaults to 0.0.
- **random_seed** (*int*) – Seed for the random number generator. Defaults to 0.

**Attributes**

| | |
|---|---|
| **hyper-parameter_ranges** | { "criterion": ["gini", "entropy"], "max_features": ["auto", "sqrt", "log2"], "max_depth": Integer(4, 10),} |
| **model_family** | ModelFamily.DECISION_TREE |
| **modifies_features** | True |
| **modifies_target** | False |
| **name** | Decision Tree Classifier |
| **supported_problem_types** | [ ProblemTypes.BINARY, ProblemTypes.MULTICLASS, ProblemTypes.TIME_SERIES_BINARY, ProblemTypes.TIME_SERIES_MULTICLASS,] |
| **training_only** | False |

**Methods**

| | |
|---|---|
| *clone* | Constructs a new component with the same parameters and random state. |
| *default_parameters* | Returns the default parameters for this component. |
| *describe* | Describe a component and its parameters. |
| *feature_importance* | Returns importance associated with each feature. |
| *fit* | Fits estimator to data. |
| *get_prediction_intervals* | Find the prediction intervals using the fitted regressor. |
| *load* | Loads component at file path. |
| *needs_fitting* | Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances. |
| *parameters* | Returns the parameters which were used to initialize the component. |
| *predict* | Make predictions using selected features. |
| *predict_proba* | Make probability estimates for labels. |
| *save* | Saves component at file path. |
| *update_parameters* | Updates the parameter dictionary of the component. |

**clone**(*self*)

Constructs a new component with the same parameters and random state.

**Returns** A new instance of this component with identical parameters and random state.

**default_parameters**(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

**Returns** Default parameters for this component.

**Return type** dict

**describe**(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

**Parameters**

- **print_name** (*bool, optional*) – whether to print name of component

- **return_dict** (*bool, optional*) – whether to return description as dictionary in the format {"name": name, "parameters": parameters}

**Returns** Returns dictionary if return_dict is True, else None.

**Return type** None or dict

**property feature_importance**(*self*) → pandas.Series

Returns importance associated with each feature.

**Returns** Importance associated with each feature.

**Return type** np.ndarray

**Raises** **MethodPropertyNotFoundError** – If estimator does not have a feature_importance method or a component_obj that implements feature_importance.

**fit**(*self, X: pandas.DataFrame, y: Optional[pandas.Series] = None*)

Fits estimator to data.

**Parameters**

- **X** (*pd.DataFrame*) – The input training data of shape [n_samples, n_features].
- **y** (*pd.Series, optional*) – The target training data of length [n_samples].

**Returns** self

**get_prediction_intervals**(*self, X: pandas.DataFrame, y: Optional[pandas.Series] = None, coverage: List[float] = None, predictions: pandas.Series = None*) → Dict[str, pandas.Series]

Find the prediction intervals using the fitted regressor.

This function takes the predictions of the fitted estimator and calculates the rolling standard deviation across all predictions using a window size of 5. The lower and upper predictions are determined by taking the percent point (quantile) function of the lower tail probability at each bound multiplied by the rolling standard deviation.

**Parameters**

- **X** (*pd.DataFrame*) – Data of shape [n_samples, n_features].
- **y** (*pd.Series*) – Target data. Ignored.
- **coverage** (*list[float]*) – A list of floats between the values 0 and 1 that the upper and lower bounds of the prediction interval should be calculated for.
- **predictions** (*pd.Series*) – Optional list of predictions to use. If None, will generate predictions using *X*.

**Returns** Prediction intervals, keys are in the format {coverage}_lower or {coverage}_upper.

**Return type** dict

**Raises** **MethodPropertyNotFoundError** – If the estimator does not support Time Series Regression as a problem type.

**static load**(*file_path*)

Loads component at file path.

**Parameters** **file_path** (*str*) – Location to load file.

**Returns** ComponentBase object

**needs_fitting**(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need to be fit or whose fit methods do nothing.

**Returns** True.

**property parameters**(*self*)

Returns the parameters which were used to initialize the component.

**predict**(*self*, *X: pandas.DataFrame*) → pandas.Series

Make predictions using selected features.

**Parameters** **X** (*pd.DataFrame*) – Data of shape [n_samples, n_features].

**Returns** Predicted values.

**Return type** pd.Series

**Raises** **MethodPropertyNotFoundError** – If estimator does not have a predict method or a component_obj that implements predict.

**predict_proba**(*self*, *X: pandas.DataFrame*) → pandas.Series

Make probability estimates for labels.

**Parameters** **X** (*pd.DataFrame*) – Features.

**Returns** Probability estimates.

**Return type** pd.Series

**Raises** **MethodPropertyNotFoundError** – If estimator does not have a predict_proba method or a component_obj that implements predict_proba.

**save**(*self*, *file_path*, *pickle_protocol=cloudpickle.DEFAULT_PROTOCOL*)

Saves component at file path.

**Parameters**

- **file_path** (*str*) – Location to save file.
- **pickle_protocol** (*int*) – The pickle data stream format.

**update_parameters**(*self*, *update_dict*, *reset_fit=True*)

Updates the parameter dictionary of the component.

**Parameters**

- **update_dict** (*dict*) – A dict of parameters to update.
- **reset_fit** (*bool, optional*) – If True, will set *_is_fitted* to False.

**class** evalml.pipelines.**DecisionTreeRegressor**(*criterion='squared_error'*, *max_features='auto'*, *max_depth=6*, *min_samples_split=2*, *min_weight_fraction_leaf=0.0*, *random_seed=0*, ***kwargs*)

Decision Tree Regressor.

**Parameters**

- **criterion** (*{"squared_error", "friedman_mse", "absolute_error", "poisson"}*) – The function to measure the quality of a split. Supported criteria are:

  - ”squared_error” for the mean squared error, which is equal to variance reduction as feature selection criterion and minimizes the L2 loss using the mean of each terminal node
  - ”friedman_mse”, which uses mean squared error with Friedman”s improvement score for potential splits
  - ”absolute_error” for the mean absolute error, which minimizes the L1 loss using the median of each terminal node,
  - ”poisson” which uses reduction in Poisson deviance to find splits.
- **max_features** (*int, float or {"auto", "sqrt", "log2"}*) – The number of features to consider when looking for the best split:
  - If int, then consider max_features features at each split.
  - If float, then max_features is a fraction and int(max_features * n_features) features are considered at each split.
  - If “auto”, then max_features=sqrt(n_features).
  - If “sqrt”, then max_features=sqrt(n_features).
  - If “log2”, then max_features=log2(n_features).
  - If None, then max_features = n_features.

  The search for a split does not stop until at least one valid partition of the node samples is found, even if it requires to effectively inspect more than max_features features.
- **max_depth** (*int*) – The maximum depth of the tree. Defaults to 6.
- **min_samples_split** (*int or float*) – The minimum number of samples required to split an internal node:
  - If int, then consider min_samples_split as the minimum number.
  - If float, then min_samples_split is a fraction and ceil(min_samples_split * n_samples) are the minimum number of samples for each split.

  Defaults to 2.
- **min_weight_fraction_leaf** (*float*) – The minimum weighted fraction of the sum total of weights (of all the input samples) required to be at a leaf node. Defaults to 0.0.
- **random_seed** (*int*) – Seed for the random number generator. Defaults to 0.

**Attributes**

| | |
|---|---|
| **hyperparameter_ranges** | { “criterion”: [“squared_error”, “friedman_mse”, “absolute_error”], “max_features”: [“auto”, “sqrt”, “log2”], “max_depth”: Integer(4, 10),} |
| **model_family** | ModelFamily.DECISION_TREE |
| **modifies_features** | True |
| **modifies_target** | False |
| **name** | Decision Tree Regressor |
| **supported_problem_types** | [ ProblemTypes.REGRESSION, ProblemTypes.TIME_SERIES_REGRESSION,] |
| **training_only** | False |

**Methods**

| | |
|---|---|
| *clone* | Constructs a new component with the same parameters and random state. |
| *default_parameters* | Returns the default parameters for this component. |
| *describe* | Describe a component and its parameters. |
| *feature_importance* | Returns importance associated with each feature. |
| *fit* | Fits estimator to data. |
| *get_prediction_intervals* | Find the prediction intervals using the fitted regressor. |
| *load* | Loads component at file path. |
| *needs_fitting* | Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances. |
| *parameters* | Returns the parameters which were used to initialize the component. |
| *predict* | Make predictions using selected features. |
| *predict_proba* | Make probability estimates for labels. |
| *save* | Saves component at file path. |
| *update_parameters* | Updates the parameter dictionary of the component. |

**clone**(*self*)

Constructs a new component with the same parameters and random state.

**Returns** A new instance of this component with identical parameters and random state.

**default_parameters**(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

**Returns** Default parameters for this component.

**Return type** dict

**describe**(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

**Parameters**

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description as dictionary in the format {"name": name, "parameters": parameters}

**Returns** Returns dictionary if return_dict is True, else None.

**Return type** None or dict

**property feature_importance**(*self*) → pandas.Series

Returns importance associated with each feature.

**Returns** Importance associated with each feature.

**Return type** np.ndarray

**Raises** **MethodPropertyNotFoundError** – If estimator does not have a feature_importance method or a component_obj that implements feature_importance.

**fit**(*self, X: pandas.DataFrame, y: Optional[pandas.Series] = None*)

Fits estimator to data.

**Parameters**

- **X** (*pd.DataFrame*) – The input training data of shape [n_samples, n_features].
- **y** (*pd.Series, optional*) – The target training data of length [n_samples].

**Returns** self

**get_prediction_intervals**(*self, X: pandas.DataFrame, y: Optional[pandas.Series] = None, coverage: List[float] = None, predictions: pandas.Series = None*) → Dict[str, pandas.Series]

Find the prediction intervals using the fitted regressor.

This function takes the predictions of the fitted estimator and calculates the rolling standard deviation across all predictions using a window size of 5. The lower and upper predictions are determined by taking the percent point (quantile) function of the lower tail probability at each bound multiplied by the rolling standard deviation.

**Parameters**

- **X** (*pd.DataFrame*) – Data of shape [n_samples, n_features].
- **y** (*pd.Series*) – Target data. Ignored.
- **coverage** (*list[float]*) – A list of floats between the values 0 and 1 that the upper and lower bounds of the prediction interval should be calculated for.
- **predictions** (*pd.Series*) – Optional list of predictions to use. If None, will generate predictions using *X*.

**Returns** Prediction intervals, keys are in the format {coverage}_lower or {coverage}_upper.

**Return type** dict

**Raises** **MethodPropertyNotFoundError** – If the estimator does not support Time Series Regression as a problem type.

**static load**(*file_path*)

Loads component at file path.

**Parameters** **file_path** (*str*) – Location to load file.

**Returns** ComponentBase object

**needs_fitting**(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need to be fit or whose fit methods do nothing.

**Returns** True.

**property parameters**(*self*)

Returns the parameters which were used to initialize the component.

**predict**(*self, X: pandas.DataFrame*) → pandas.Series

Make predictions using selected features.

**Parameters** **X** (*pd.DataFrame*) – Data of shape [n_samples, n_features].

**Returns** Predicted values.

**Return type** pd.Series

**Raises** `MethodPropertyNotFoundError` – If estimator does not have a predict method or a component_obj that implements predict.

`predict_proba`(*self*, *X: pandas.DataFrame*) → pandas.Series

Make probability estimates for labels.

**Parameters** `X` (*pd.DataFrame*) – Features.

**Returns** Probability estimates.

**Return type** pd.Series

**Raises** `MethodPropertyNotFoundError` – If estimator does not have a predict_proba method or a component_obj that implements predict_proba.

`save`(*self*, *file_path*, *pickle_protocol=cloudpickle.DEFAULT_PROTOCOL*)

Saves component at file path.

**Parameters**

- `file_path` (*str*) – Location to save file.
- `pickle_protocol` (*int*) – The pickle data stream format.

`update_parameters`(*self*, *update_dict*, *reset_fit=True*)

Updates the parameter dictionary of the component.

**Parameters**

- `update_dict` (*dict*) – A dict of parameters to update.
- `reset_fit` (*bool, optional*) – If True, will set *_is_fitted* to False.

class `evalml.pipelines.DFSTransformer`(*index='index'*, *features=None*, *random_seed=0*, ***kwargs*)

Featuretools DFS component that generates features for the input features.

**Parameters**

- `index` (*string*) – The name of the column that contains the indices. If no column with this name exists, then featuretools.EntitySet() creates a column with this name to serve as the index column. Defaults to 'index'.
- `random_seed` (*int*) – Seed for the random number generator. Defaults to 0.
- `features` (*list*) – List of features to run DFS on. Defaults to None. Features will only be computed if the columns used by the feature exist in the input and if the feature itself is not in input. If features is an empty list, no transformation will occur to inputted data.

**Attributes**

| | |
|---|---|
| **hyper-parameter_ranges** | {} |
| **modifies_features** | True |
| **modifies_target** | False |
| **name** | DFS Transformer |
| **training_only** | False |

**Methods**

| | |
|---|---|
| *clone* | Constructs a new component with the same parameters and random state. |
| *contains_pre_existing_features* | Determines whether or not features from a DFS Transformer match pipeline input features. |
| *default_parameters* | Returns the default parameters for this component. |
| *describe* | Describe a component and its parameters. |
| *fit* | Fits the DFSTransformer Transformer component. |
| *fit_transform* | Fits on X and transforms X. |
| *load* | Loads component at file path. |
| *needs_fitting* | Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances. |
| *parameters* | Returns the parameters which were used to initialize the component. |
| *save* | Saves component at file path. |
| *transform* | Computes the feature matrix for the input X using featuretools' dfs algorithm. |
| *update_parameters* | Updates the parameter dictionary of the component. |

**clone**(*self*)

Constructs a new component with the same parameters and random state.

**Returns** A new instance of this component with identical parameters and random state.

**static contains_pre_existing_features**(*dfs_features: Optional[List[featuretools.feature_base.FeatureBase]], input_feature_names: List[str], target: Optional[str] = None*)

Determines whether or not features from a DFS Transformer match pipeline input features.

**Parameters**

- **dfs_features** (*Optional[List[FeatureBase]]*) – List of features output from a DFS Transformer.
- **input_feature_names** (*List[str]*) – List of input features into the DFS Transformer.
- **target** (*Optional[str]*) – The target whose values we are trying to predict. This is used to know which column to ignore if the target column is present in the list of features in the DFS Transformer's parameters.

**default_parameters**(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

**Returns** Default parameters for this component.

**Return type** dict

**describe**(*self, print_name=False, return_dict=False*)

Describe a component and its parameters.

**Parameters**

- **print_name** (*bool, optional*) – whether to print name of component

- **return_dict** (*bool, optional*) – whether to return description as dictionary in the format {"name": name, "parameters": parameters}

**Returns** Returns dictionary if return_dict is True, else None.

**Return type** None or dict

**fit**(*self*, *X*, *y=None*)

Fits the DFSTransformer Transformer component.

**Parameters**

- **X** (*pd.DataFrame, np.array*) – The input data to transform, of shape [n_samples, n_features].
- **y** (*pd.Series*) – The target training data of length [n_samples].

**Returns** self

**fit_transform**(*self*, *X*, *y=None*)

Fits on X and transforms X.

**Parameters**

- **X** (*pd.DataFrame*) – Data to fit and transform.
- **y** (*pd.Series*) – Target data.

**Returns** Transformed X.

**Return type** pd.DataFrame

**Raises** **MethodPropertyNotFoundError** – If transformer does not have a transform method or a component_obj that implements transform.

**static load**(*file_path*)

Loads component at file path.

**Parameters** **file_path** (*str*) – Location to load file.

**Returns** ComponentBase object

**needs_fitting**(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need to be fit or whose fit methods do nothing.

**Returns** True.

**property parameters**(*self*)

Returns the parameters which were used to initialize the component.

**save**(*self*, *file_path*, *pickle_protocol=cloudpickle.DEFAULT_PROTOCOL*)

Saves component at file path.

**Parameters**

- **file_path** (*str*) – Location to save file.
- **pickle_protocol** (*int*) – The pickle data stream format.

**transform**(*self*, *X*, *y=None*)

Computes the feature matrix for the input X using featuretools' dfs algorithm.

**Parameters**

- **X** (*pd.DataFrame or np.ndarray*) – The input training data to transform. Has shape [n_samples, n_features]
- **y** (*pd.Series, optional*) – Ignored.

**Returns** Feature matrix

**Return type** pd.DataFrame

**update_parameters**(*self*, *update_dict*, *reset_fit=True*)

Updates the parameter dictionary of the component.

**Parameters**

- **update_dict** (*dict*) – A dict of parameters to update.
- **reset_fit** (*bool, optional*) – If True, will set *_is_fitted* to False.

**class** evalml.pipelines.**DropNaNRowsTransformer**(*parameters=None*, *component_obj=None*, *random_seed=0*, ***kwargs*)

Transformer to drop rows with NaN values.

**Parameters** **random_seed** (*int*) – Seed for the random number generator. Is not used by this component. Defaults to 0.

**Attributes**

| | |
|---|---|
| **hyper-parameter_ranges** | {} |
| **modifies_features** | True |
| **modifies_target** | True |
| **name** | Drop NaN Rows Transformer |
| **training_only** | False |

**Methods**

| | |
|---|---|
| *clone* | Constructs a new component with the same parameters and random state. |
| *default_parameters* | Returns the default parameters for this component. |
| *describe* | Describe a component and its parameters. |
| *fit* | Fits component to data. |
| *fit_transform* | Fits on X and transforms X. |
| *load* | Loads component at file path. |
| *needs_fitting* | Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances. |
| *parameters* | Returns the parameters which were used to initialize the component. |
| *save* | Saves component at file path. |
| *transform* | Transforms data using fitted component. |
| *update_parameters* | Updates the parameter dictionary of the component. |

**clone**(*self*)

Constructs a new component with the same parameters and random state.

**Returns** A new instance of this component with identical parameters and random state.

**default_parameters**(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

**Returns** Default parameters for this component.

**Return type** dict

**describe**(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

**Parameters**

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description as dictionary in the format {"name": name, "parameters": parameters}

**Returns** Returns dictionary if return_dict is True, else None.

**Return type** None or dict

**fit**(*self*, *X*, *y=None*)

Fits component to data.

**Parameters**

- **X** (*pd.DataFrame*) – The input training data of shape [n_samples, n_features].
- **y** (*pd.Series, optional*) – The target training data of length [n_samples].

**Returns** self

**fit_transform**(*self*, *X*, *y=None*)

Fits on X and transforms X.

**Parameters**

- **X** (*pd.DataFrame*) – Data to fit and transform.

- **y** (*pd.Series*) – Target data.

**Returns** Transformed X.

**Return type** pd.DataFrame

**Raises** **MethodPropertyNotFoundError** – If transformer does not have a transform method or a component_obj that implements transform.

**static load**(*file_path*)

Loads component at file path.

**Parameters** **file_path** (*str*) – Location to load file.

**Returns** ComponentBase object

**needs_fitting**(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need to be fit or whose fit methods do nothing.

**Returns** True.

**property parameters**(*self*)

Returns the parameters which were used to initialize the component.

**save**(*self*, *file_path*, *pickle_protocol=cloudpickle.DEFAULT_PROTOCOL*)

Saves component at file path.

**Parameters**

- **file_path** (*str*) – Location to save file.
- **pickle_protocol** (*int*) – The pickle data stream format.

**transform**(*self*, *X*, *y=None*)

Transforms data using fitted component.

**Parameters**

- **X** (*pd.DataFrame*) – Features.
- **y** (*pd.Series, optional*) – Target data.

**Returns** Data with NaN rows dropped.

**Return type** (pd.DataFrame, pd.Series)

**update_parameters**(*self*, *update_dict*, *reset_fit=True*)

Updates the parameter dictionary of the component.

**Parameters**

- **update_dict** (*dict*) – A dict of parameters to update.
- **reset_fit** (*bool, optional*) – If True, will set *_is_fitted* to False.

**class** evalml.pipelines.**ElasticNetClassifier**(*penalty='elasticnet'*, *C=1.0*, *l1_ratio=0.15*, *multi_class='auto'*, *solver='saga'*, *n_jobs=- 1*, *random_seed=0*, ***kwargs*)

Elastic Net Classifier. Uses Logistic Regression with elasticnet penalty as the base estimator.

**Parameters**

- **penalty** (*{"l1", "l2", "elasticnet", "none"}*) – The norm used in penalization. Defaults to "elasticnet".
- **C** (*float*) – Inverse of regularization strength. Must be a positive float. Defaults to 1.0.
- **l1_ratio** (*float*) – The mixing parameter, with 0 <= l1_ratio <= 1. Only used if penalty='elasticnet'. Setting l1_ratio=0 is equivalent to using penalty='l2', while setting l1_ratio=1 is equivalent to using penalty='l1'. For 0 < l1_ratio <1, the penalty is a combination of L1 and L2. Defaults to 0.15.
- **multi_class** (*{"auto", "ovr", "multinomial"}*) – If the option chosen is "ovr", then a binary problem is fit for each label. For "multinomial" the loss minimised is the multinomial loss fit across the entire probability distribution, even when the data is binary. "multinomial" is unavailable when solver="liblinear". "auto" selects "ovr" if the data is binary, or if solver="liblinear", and otherwise selects "multinomial". Defaults to "auto".
- **solver** (*{"newton-cg", "lbfgs", "liblinear", "sag", "saga"}*) – Algorithm to use in the optimization problem. For small datasets, "liblinear" is a good choice, whereas "sag" and "saga" are faster for large ones. For multiclass problems, only "newton-cg", "sag", "saga" and "lbfgs" handle multinomial loss; "liblinear" is limited to one-versus-rest schemes.
  - "newton-cg", "lbfgs", "sag" and "saga" handle L2 or no penalty
  - "liblinear" and "saga" also handle L1 penalty
  - "saga" also supports "elasticnet" penalty
  - "liblinear" does not support setting penalty='none'

  Defaults to "saga".
- **n_jobs** (*int*) – Number of parallel threads used to run xgboost. Note that creating thread contention will significantly slow down the algorithm. Defaults to -1.
- **random_seed** (*int*) – Seed for the random number generator. Defaults to 0.

**Attributes**

| | |
|---|---|
| **hyper-parameter_ranges** | { "C": Real(0.01, 10), "l1_ratio": Real(0, 1)} |
| **model_family** | ModelFamily.LINEAR_MODEL |
| **modifies_features** | True |
| **modifies_target** | False |
| **name** | Elastic Net Classifier |
| **supported_problem_types** | [ ProblemTypes.BINARY, ProblemTypes.MULTICLASS, ProblemTypes.TIME_SERIES_BINARY, ProblemTypes.TIME_SERIES_MULTICLASS,] |
| **training_only** | False |

**Methods**

| | |
|---|---|
| *clone* | Constructs a new component with the same parameters and random state. |
| *default_parameters* | Returns the default parameters for this component. |
| *describe* | Describe a component and its parameters. |
| *feature_importance* | Feature importance for fitted ElasticNet classifier. |
| *fit* | Fits ElasticNet classifier component to data. |
| *get_prediction_intervals* | Find the prediction intervals using the fitted regressor. |
| *load* | Loads component at file path. |
| *needs_fitting* | Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances. |
| *parameters* | Returns the parameters which were used to initialize the component. |
| *predict* | Make predictions using selected features. |
| *predict_proba* | Make probability estimates for labels. |
| *save* | Saves component at file path. |
| *update_parameters* | Updates the parameter dictionary of the component. |

**clone**(*self*)

Constructs a new component with the same parameters and random state.

**Returns** A new instance of this component with identical parameters and random state.

**default_parameters**(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

**Returns** Default parameters for this component.

**Return type** dict

**describe**(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

**Parameters**

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description as dictionary in the format {"name": name, "parameters": parameters}

**Returns** Returns dictionary if return_dict is True, else None.

**Return type** None or dict

**property feature_importance**(*self*)

Feature importance for fitted ElasticNet classifier.

**fit**(*self*, *X*, *y*)

Fits ElasticNet classifier component to data.

**Parameters**

- **X** (*pd.DataFrame*) – The input training data of shape [n_samples, n_features].
- **y** (*pd.Series*) – The target training data of length [n_samples].

**Returns** self

**get_prediction_intervals**(*self*, *X: pandas.DataFrame*, *y: Optional[pandas.Series] = None*, *coverage: List[float] = None*, *predictions: pandas.Series = None*) → Dict[str, pandas.Series]

Find the prediction intervals using the fitted regressor.

This function takes the predictions of the fitted estimator and calculates the rolling standard deviation across all predictions using a window size of 5. The lower and upper predictions are determined by taking the percent point (quantile) function of the lower tail probability at each bound multiplied by the rolling standard deviation.

**Parameters**

- **X** (*pd.DataFrame*) – Data of shape [n_samples, n_features].
- **y** (*pd.Series*) – Target data. Ignored.
- **coverage** (*list[float]*) – A list of floats between the values 0 and 1 that the upper and lower bounds of the prediction interval should be calculated for.
- **predictions** (*pd.Series*) – Optional list of predictions to use. If None, will generate predictions using *X*.

**Returns** Prediction intervals, keys are in the format {coverage}_lower or {coverage}_upper.

**Return type** dict

**Raises** **MethodPropertyNotFoundError** – If the estimator does not support Time Series Regression as a problem type.

**static load**(*file_path*)

Loads component at file path.

**Parameters** **file_path** (*str*) – Location to load file.

**Returns** ComponentBase object

**needs_fitting**(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need to be fit or whose fit methods do nothing.

**Returns** True.

**property parameters**(*self*)

Returns the parameters which were used to initialize the component.

**predict**(*self*, *X: pandas.DataFrame*) → pandas.Series

Make predictions using selected features.

**Parameters** **X** (*pd.DataFrame*) – Data of shape [n_samples, n_features].

**Returns** Predicted values.

**Return type** pd.Series

**Raises** **MethodPropertyNotFoundError** – If estimator does not have a predict method or a component_obj that implements predict.

**predict_proba**(*self*, *X: pandas.DataFrame*) → pandas.Series

Make probability estimates for labels.

**Parameters** **X** (*pd.DataFrame*) – Features.

**Returns** Probability estimates.

**Return type** pd.Series

**Raises** `MethodPropertyNotFoundError` – If estimator does not have a predict_proba method or a component_obj that implements predict_proba.

`save`(*self*, *file_path*, *pickle_protocol=cloudpickle.DEFAULT_PROTOCOL*)

Saves component at file path.

**Parameters**

- `file_path` (*str*) – Location to save file.
- `pickle_protocol` (*int*) – The pickle data stream format.

`update_parameters`(*self*, *update_dict*, *reset_fit=True*)

Updates the parameter dictionary of the component.

**Parameters**

- `update_dict` (*dict*) – A dict of parameters to update.
- `reset_fit` (*bool, optional*) – If True, will set *_is_fitted* to False.

**class** `evalml.pipelines.`**`ElasticNetRegressor`**(*alpha=0.0001*, *l1_ratio=0.15*, *max_iter=1000*, *random_seed=0*, ***kwargs*)

Elastic Net Regressor.

**Parameters**

- `alpha` (*float*) – Constant that multiplies the penalty terms. Defaults to 0.0001.
- `l1_ratio` (*float*) – The mixing parameter, with 0 <= l1_ratio <= 1. Only used if penalty='elasticnet'. Setting l1_ratio=0 is equivalent to using penalty='l2', while setting l1_ratio=1 is equivalent to using penalty='l1'. For 0 < l1_ratio <1, the penalty is a combination of L1 and L2. Defaults to 0.15.
- `max_iter` (*int*) – The maximum number of iterations. Defaults to 1000.
- `random_seed` (*int*) – Seed for the random number generator. Defaults to 0.

**Attributes**

| | |
|---|---|
| **hyper-parameter_ranges** | { "alpha": Real(0, 1), "l1_ratio": Real(0, 1),} |
| **model_family** | ModelFamily.LINEAR_MODEL |
| **modifies_features** | True |
| **modifies_target** | False |
| **name** | Elastic Net Regressor |
| **supported_problem_types** | [ ProblemTypes.REGRESSION, ProblemTypes.TIME_SERIES_REGRESSION,] |
| **training_only** | False |

**Methods**

| | |
|---|---|
| *clone* | Constructs a new component with the same parameters and random state. |
| *default_parameters* | Returns the default parameters for this component. |
| *describe* | Describe a component and its parameters. |
| *feature_importance* | Feature importance for fitted ElasticNet regressor. |
| *fit* | Fits estimator to data. |
| *get_prediction_intervals* | Find the prediction intervals using the fitted regressor. |
| *load* | Loads component at file path. |
| *needs_fitting* | Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances. |
| *parameters* | Returns the parameters which were used to initialize the component. |
| *predict* | Make predictions using selected features. |
| *predict_proba* | Make probability estimates for labels. |
| *save* | Saves component at file path. |
| *update_parameters* | Updates the parameter dictionary of the component. |

**clone**(*self*)

Constructs a new component with the same parameters and random state.

**Returns** A new instance of this component with identical parameters and random state.

**default_parameters**(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

**Returns** Default parameters for this component.

**Return type** dict

**describe**(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

**Parameters**

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description as dictionary in the format {"name": name, "parameters": parameters}

**Returns** Returns dictionary if return_dict is True, else None.

**Return type** None or dict

**property feature_importance**(*self*)

Feature importance for fitted ElasticNet regressor.

**fit**(*self*, *X: pandas.DataFrame*, *y: Optional[pandas.Series] = None*)

Fits estimator to data.

**Parameters**

- **X** (*pd.DataFrame*) – The input training data of shape [n_samples, n_features].
- **y** (*pd.Series, optional*) – The target training data of length [n_samples].

**Returns** self

**get_prediction_intervals**(*self*, *X: pandas.DataFrame*, *y: Optional[pandas.Series] = None*, *coverage: List[float] = None*, *predictions: pandas.Series = None*) → Dict[str, pandas.Series]

Find the prediction intervals using the fitted regressor.

This function takes the predictions of the fitted estimator and calculates the rolling standard deviation across all predictions using a window size of 5. The lower and upper predictions are determined by taking the percent point (quantile) function of the lower tail probability at each bound multiplied by the rolling standard deviation.

**Parameters**

- **X** (*pd.DataFrame*) – Data of shape [n_samples, n_features].
- **y** (*pd.Series*) – Target data. Ignored.
- **coverage** (*list[float]*) – A list of floats between the values 0 and 1 that the upper and lower bounds of the prediction interval should be calculated for.
- **predictions** (*pd.Series*) – Optional list of predictions to use. If None, will generate predictions using *X*.

**Returns** Prediction intervals, keys are in the format {coverage}_lower or {coverage}_upper.

**Return type** dict

**Raises** **MethodPropertyNotFoundError** – If the estimator does not support Time Series Regression as a problem type.

**static load**(*file_path*)

Loads component at file path.

**Parameters** **file_path** (*str*) – Location to load file.

**Returns** ComponentBase object

**needs_fitting**(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need to be fit or whose fit methods do nothing.

**Returns** True.

**property parameters**(*self*)

Returns the parameters which were used to initialize the component.

**predict**(*self*, *X: pandas.DataFrame*) → pandas.Series

Make predictions using selected features.

**Parameters** **X** (*pd.DataFrame*) – Data of shape [n_samples, n_features].

**Returns** Predicted values.

**Return type** pd.Series

**Raises** **MethodPropertyNotFoundError** – If estimator does not have a predict method or a component_obj that implements predict.

**predict_proba**(*self*, *X: pandas.DataFrame*) → pandas.Series

Make probability estimates for labels.

**Parameters** **X** (*pd.DataFrame*) – Features.

**Returns** Probability estimates.

**Return type** pd.Series

**Raises** **MethodPropertyNotFoundError** – If estimator does not have a predict_proba method or a component_obj that implements predict_proba.

**save**(*self*, *file_path*, *pickle_protocol=cloudpickle.DEFAULT_PROTOCOL*)

Saves component at file path.

**Parameters**

- **file_path** (*str*) – Location to save file.
- **pickle_protocol** (*int*) – The pickle data stream format.

**update_parameters**(*self*, *update_dict*, *reset_fit=True*)

Updates the parameter dictionary of the component.

**Parameters**

- **update_dict** (*dict*) – A dict of parameters to update.
- **reset_fit** (*bool, optional*) – If True, will set *_is_fitted* to False.

**class** evalml.pipelines.**Estimator**(*parameters: dict = None*, *component_obj: Type[*evalml.pipelines.components.ComponentBase*] = None*, *random_seed: Union[int, float] = 0*, ***kwargs*)

A component that fits and predicts given data.

To implement a new Estimator, define your own class which is a subclass of Estimator, including a name and a list of acceptable ranges for any parameters to be tuned during the automl search (hyperparameters). Define an *__init__* method which sets up any necessary state and objects. Make sure your *__init__* only uses standard keyword arguments and calls *super().__init__()* with a parameters dict. You may also override the *fit*, *transform*, *fit_transform* and other methods in this class if appropriate.

To see some examples, check out the definitions of any Estimator component subclass.

**Parameters**

- **parameters** (*dict*) – Dictionary of parameters for the component. Defaults to None.
- **component_obj** (*obj*) – Third-party objects useful in component implementation. Defaults to None.
- **random_seed** (*int*) – Seed for the random number generator. Defaults to 0.

**Attributes**

| | |
|---|---|
| **model_family** | ModelFamily.NONE |
| **modifies_features** | True |
| **modifies_target** | False |
| **training_only** | False |

**Methods**

| | |
|---|---|
| *clone* | Constructs a new component with the same parameters and random state. |
| *default_parameters* | Returns the default parameters for this component. |
| *describe* | Describe a component and its parameters. |
| *feature_importance* | Returns importance associated with each feature. |
| *fit* | Fits estimator to data. |
| *get_prediction_intervals* | Find the prediction intervals using the fitted regressor. |
| *load* | Loads component at file path. |
| *model_family* | ModelFamily.NONE |
| *name* | Returns string name of this component. |
| *needs_fitting* | Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances. |
| *parameters* | Returns the parameters which were used to initialize the component. |
| *predict* | Make predictions using selected features. |
| *predict_proba* | Make probability estimates for labels. |
| *save* | Saves component at file path. |
| *supported_problem_types* | Problem types this estimator supports. |
| *update_parameters* | Updates the parameter dictionary of the component. |

**clone**(*self*)

Constructs a new component with the same parameters and random state.

**Returns** A new instance of this component with identical parameters and random state.

**default_parameters**(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

**Returns** Default parameters for this component.

**Return type** dict

**describe**(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

**Parameters**

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description as dictionary in the format {"name": name, "parameters": parameters}

**Returns** Returns dictionary if return_dict is True, else None.

**Return type** None or dict

**property feature_importance**(*self*) → pandas.Series

Returns importance associated with each feature.

**Returns** Importance associated with each feature.

**Return type** np.ndarray

**Raises** **MethodPropertyNotFoundError** – If estimator does not have a feature_importance method or a component_obj that implements feature_importance.

**fit**(*self*, *X: pandas.DataFrame*, *y: Optional[pandas.Series] = None*)

Fits estimator to data.

**Parameters**

- **X** (*pd.DataFrame*) – The input training data of shape [n_samples, n_features].
- **y** (*pd.Series, optional*) – The target training data of length [n_samples].

**Returns** self

**get_prediction_intervals**(*self*, *X: pandas.DataFrame*, *y: Optional[pandas.Series] = None*, *coverage: List[float] = None*, *predictions: pandas.Series = None*) → Dict[str, pandas.Series]

Find the prediction intervals using the fitted regressor.

This function takes the predictions of the fitted estimator and calculates the rolling standard deviation across all predictions using a window size of 5. The lower and upper predictions are determined by taking the percent point (quantile) function of the lower tail probability at each bound multiplied by the rolling standard deviation.

**Parameters**

- **X** (*pd.DataFrame*) – Data of shape [n_samples, n_features].
- **y** (*pd.Series*) – Target data. Ignored.
- **coverage** (*list[float]*) – A list of floats between the values 0 and 1 that the upper and lower bounds of the prediction interval should be calculated for.
- **predictions** (*pd.Series*) – Optional list of predictions to use. If None, will generate predictions using *X*.

**Returns** Prediction intervals, keys are in the format {coverage}_lower or {coverage}_upper.

**Return type** dict

**Raises** **MethodPropertyNotFoundError** – If the estimator does not support Time Series Regression as a problem type.

**static load**(*file_path*)

Loads component at file path.

**Parameters** **file_path** (*str*) – Location to load file.

**Returns** ComponentBase object

**property model_family**(*cls*)

Returns ModelFamily of this component.

**property name**(*cls*)

Returns string name of this component.

**needs_fitting**(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need to be fit or whose fit methods do nothing.

**Returns** True.

**property parameters**(*self*)

Returns the parameters which were used to initialize the component.

**predict**(*self*, *X: pandas.DataFrame*) → pandas.Series

Make predictions using selected features.

**Parameters** **X** (*pd.DataFrame*) – Data of shape [n_samples, n_features].

**Returns** Predicted values.

**Return type** pd.Series

**Raises** **MethodPropertyNotFoundError** – If estimator does not have a predict method or a component_obj that implements predict.

**predict_proba**(*self*, *X: pandas.DataFrame*) → pandas.Series

Make probability estimates for labels.

**Parameters** **X** (*pd.DataFrame*) – Features.

**Returns** Probability estimates.

**Return type** pd.Series

**Raises** **MethodPropertyNotFoundError** – If estimator does not have a predict_proba method or a component_obj that implements predict_proba.

**save**(*self*, *file_path*, *pickle_protocol=cloudpickle.DEFAULT_PROTOCOL*)

Saves component at file path.

**Parameters**

- **file_path** (*str*) – Location to save file.
- **pickle_protocol** (*int*) – The pickle data stream format.

**property supported_problem_types**(*cls*)

Problem types this estimator supports.

**update_parameters**(*self*, *update_dict*, *reset_fit=True*)

Updates the parameter dictionary of the component.

**Parameters**

- **update_dict** (*dict*) – A dict of parameters to update.
- **reset_fit** (*bool, optional*) – If True, will set *_is_fitted* to False.

**class** evalml.pipelines.**ExponentialSmoothingRegressor**(*trend: Optional[str] = None*, *damped_trend: bool = False*, *seasonal: Optional[str] = None*, *sp: int = 2*, *n_jobs: int = - 1*, *random_seed: Union[int, float] = 0*, ***kwargs*)

Holt-Winters Exponential Smoothing Forecaster.

Currently ExponentialSmoothingRegressor isn't supported via conda install. It's recommended that it be installed via PyPI.

**Parameters**

- **trend** (*str*) – Type of trend component. Defaults to None.
- **damped_trend** (*bool*) – If the trend component should be damped. Defaults to False.
- **seasonal** (*str*) – Type of seasonal component. Takes one of {"additive", None}. Can also be multiplicative if
- **0** (*none of the target data is*) –

- **None.** (*but AutoMLSearch wiill not tune for this. Defaults to*) –
- **sp** (*int*) – The number of seasonal periods to consider. Defaults to 2.
- **n_jobs** (*int or None*) – Non-negative integer describing level of parallelism used for pipelines. Defaults to -1.
- **random_seed** (*int*) – Seed for the random number generator. Defaults to 0.

**Attributes**

| | |
|---|---|
| **hyper-parameter_ranges** | { "trend": [None, "additive"], "damped_trend": [True, False], "seasonal": [None, "additive"], "sp": Integer(2, 8),} |
| **model_family** | ModelFamily.EXPONENTIAL_SMOOTHING |
| **modifies_features** | True |
| **modifies_target** | False |
| **name** | Exponential Smoothing Regressor |
| **supported_problem_types** | [ProblemTypes.TIME_SERIES_REGRESSION] |
| **training_only** | False |

**Methods**

| | |
|---|---|
| *clone* | Constructs a new component with the same parameters and random state. |
| *default_parameters* | Returns the default parameters for this component. |
| *describe* | Describe a component and its parameters. |
| *feature_importance* | Returns array of 0's with a length of 1 as feature_importance is not defined for Exponential Smoothing regressor. |
| *fit* | Fits Exponential Smoothing Regressor to data. |
| *get_prediction_intervals* | Find the prediction intervals using the fitted ExponentialSmoothingRegressor. |
| *load* | Loads component at file path. |
| *needs_fitting* | Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances. |
| *parameters* | Returns the parameters which were used to initialize the component. |
| *predict* | Make predictions using fitted Exponential Smoothing regressor. |
| *predict_proba* | Make probability estimates for labels. |
| *save* | Saves component at file path. |
| *update_parameters* | Updates the parameter dictionary of the component. |

**clone**(*self*)

Constructs a new component with the same parameters and random state.

**Returns** A new instance of this component with identical parameters and random state.

**default_parameters**(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

**Returns** Default parameters for this component.

**Return type** dict

**describe**(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

**Parameters**

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description as dictionary in the format {"name": name, "parameters": parameters}

**Returns** Returns dictionary if return_dict is True, else None.

**Return type** None or dict

**property feature_importance**(*self*) → pandas.Series

Returns array of 0's with a length of 1 as feature_importance is not defined for Exponential Smoothing regressor.

**fit**(*self*, *X: pandas.DataFrame*, *y: Optional[pandas.Series] = None*)

Fits Exponential Smoothing Regressor to data.

**Parameters**

- **X** (*pd.DataFrame*) – The input training data of shape [n_samples, n_features]. Ignored.
- **y** (*pd.Series*) – The target training data of length [n_samples].

**Returns** self

**Raises** **ValueError** – If y was not passed in.

**get_prediction_intervals**(*self*, *X: pandas.DataFrame*, *y: Optional[pandas.Series] = None*, *coverage: List[float] = None*, *predictions: pandas.Series = None*) → Dict[str, pandas.Series]

Find the prediction intervals using the fitted ExponentialSmoothingRegressor.

Calculates the prediction intervals by using a simulation of the time series following a specified state space model.

**Parameters**

- **X** (*pd.DataFrame*) – Data of shape [n_samples, n_features].
- **y** (*pd.Series*) – Target data. Optional.
- **coverage** (*List[float]*) – A list of floats between the values 0 and 1 that the upper and lower bounds of the prediction interval should be calculated for.
- **predictions** (*pd.Series*) – Not used for Exponential Smoothing regressor.

**Returns** Prediction intervals, keys are in the format {coverage}_lower or {coverage}_upper.

**Return type** dict

**static load**(*file_path*)

Loads component at file path.

**Parameters** **file_path** (*str*) – Location to load file.

**Returns** ComponentBase object

**needs_fitting**(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need to be fit or whose fit methods do nothing.

**Returns** True.

**property parameters**(*self*)

Returns the parameters which were used to initialize the component.

**predict**(*self*, *X: pandas.DataFrame*, *y: Optional[pandas.Series] = None*) → pandas.Series

Make predictions using fitted Exponential Smoothing regressor.

**Parameters**

- **X** (*pd.DataFrame*) – Data of shape [n_samples, n_features]. Ignored except to set forecast horizon.
- **y** (*pd.Series*) – Target data.

**Returns** Predicted values.

**Return type** pd.Series

**predict_proba**(*self*, *X: pandas.DataFrame*) → pandas.Series

Make probability estimates for labels.

**Parameters** **X** (*pd.DataFrame*) – Features.

**Returns** Probability estimates.

**Return type** pd.Series

**Raises** **MethodPropertyNotFoundError** – If estimator does not have a predict_proba method or a component_obj that implements predict_proba.

**save**(*self*, *file_path*, *pickle_protocol=cloudpickle.DEFAULT_PROTOCOL*)

Saves component at file path.

**Parameters**

- **file_path** (*str*) – Location to save file.
- **pickle_protocol** (*int*) – The pickle data stream format.

**update_parameters**(*self*, *update_dict*, *reset_fit=True*)

Updates the parameter dictionary of the component.

**Parameters**

- **update_dict** (*dict*) – A dict of parameters to update.
- **reset_fit** (*bool, optional*) – If True, will set *_is_fitted* to False.

**class** evalml.pipelines.**ExtraTreesClassifier**(*n_estimators=100*, *max_features='auto'*, *max_depth=6*, *min_samples_split=2*, *min_weight_fraction_leaf=0.0*, *n_jobs=- 1*, *random_seed=0*, ***kwargs*)

Extra Trees Classifier.

**Parameters**

- **n_estimators** (*float*) – The number of trees in the forest. Defaults to 100.
- **max_features** (*int, float or {"auto", "sqrt", "log2"}*) – The number of features to consider when looking for the best split:
  - If int, then consider max_features features at each split.
  - If float, then max_features is a fraction and int(max_features * n_features) features are considered at each split.
  - If "auto", then max_features=sqrt(n_features).
  - If "sqrt", then max_features=sqrt(n_features).
  - If "log2", then max_features=log2(n_features).
  - If None, then max_features = n_features.

  The search for a split does not stop until at least one valid partition of the node samples is found, even if it requires to effectively inspect more than max_features features. Defaults to "auto".
- **max_depth** (*int*) – The maximum depth of the tree. Defaults to 6.
- **min_samples_split** (*int or float*) – The minimum number of samples required to split an internal node:
  - If int, then consider min_samples_split as the minimum number.
  - If float, then min_samples_split is a fraction and ceil(min_samples_split * n_samples) are the minimum number of samples for each split.
- **2.** (*Defaults to*) –
- **min_weight_fraction_leaf** (*float*) – The minimum weighted fraction of the sum total of weights (of all the input samples) required to be at a leaf node. Defaults to 0.0.
- **n_jobs** (*int or None*) – Number of jobs to run in parallel. -1 uses all processes. Defaults to -1.
- **random_seed** (*int*) – Seed for the random number generator. Defaults to 0.

**Attributes**

| | |
|---|---|
| **hyper-parameter_ranges** | { "n_estimators": Integer(10, 1000), "max_features": ["auto", "sqrt", "log2"], "max_depth": Integer(4, 10),} |
| **model_family** | ModelFamily.EXTRA_TREES |
| **modifies_features** | True |
| **modifies_target** | False |
| **name** | Extra Trees Classifier |
| **supported_problem_types** | [ ProblemTypes.BINARY, ProblemTypes.MULTICLASS, ProblemTypes.TIME_SERIES_BINARY, ProblemTypes.TIME_SERIES_MULTICLASS,] |
| **training_only** | False |

**Methods**

| | |
|---|---|
| *clone* | Constructs a new component with the same parameters and random state. |
| *default_parameters* | Returns the default parameters for this component. |
| *describe* | Describe a component and its parameters. |
| *feature_importance* | Returns importance associated with each feature. |
| *fit* | Fits estimator to data. |
| *get_prediction_intervals* | Find the prediction intervals using the fitted regressor. |
| *load* | Loads component at file path. |
| *needs_fitting* | Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances. |
| *parameters* | Returns the parameters which were used to initialize the component. |
| *predict* | Make predictions using selected features. |
| *predict_proba* | Make probability estimates for labels. |
| *save* | Saves component at file path. |
| *update_parameters* | Updates the parameter dictionary of the component. |

**clone**(*self*)

Constructs a new component with the same parameters and random state.

**Returns** A new instance of this component with identical parameters and random state.

**default_parameters**(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

**Returns** Default parameters for this component.

**Return type** dict

**describe**(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

**Parameters**

- **print_name** (*bool, optional*) – whether to print name of component

- **return_dict** (*bool, optional*) – whether to return description as dictionary in the format {"name": name, "parameters": parameters}

**Returns** Returns dictionary if return_dict is True, else None.

**Return type** None or dict

**property feature_importance**(*self*) → pandas.Series

Returns importance associated with each feature.

**Returns** Importance associated with each feature.

**Return type** np.ndarray

**Raises** **MethodPropertyNotFoundError** – If estimator does not have a feature_importance method or a component_obj that implements feature_importance.

**fit**(*self, X: pandas.DataFrame, y: Optional[pandas.Series] = None*)

Fits estimator to data.

**Parameters**

- **X** (*pd.DataFrame*) – The input training data of shape [n_samples, n_features].
- **y** (*pd.Series, optional*) – The target training data of length [n_samples].

**Returns** self

**get_prediction_intervals**(*self, X: pandas.DataFrame, y: Optional[pandas.Series] = None, coverage: List[float] = None, predictions: pandas.Series = None*) → Dict[str, pandas.Series]

Find the prediction intervals using the fitted regressor.

This function takes the predictions of the fitted estimator and calculates the rolling standard deviation across all predictions using a window size of 5. The lower and upper predictions are determined by taking the percent point (quantile) function of the lower tail probability at each bound multiplied by the rolling standard deviation.

**Parameters**

- **X** (*pd.DataFrame*) – Data of shape [n_samples, n_features].
- **y** (*pd.Series*) – Target data. Ignored.
- **coverage** (*list[float]*) – A list of floats between the values 0 and 1 that the upper and lower bounds of the prediction interval should be calculated for.
- **predictions** (*pd.Series*) – Optional list of predictions to use. If None, will generate predictions using *X*.

**Returns** Prediction intervals, keys are in the format {coverage}_lower or {coverage}_upper.

**Return type** dict

**Raises** **MethodPropertyNotFoundError** – If the estimator does not support Time Series Regression as a problem type.

**static load**(*file_path*)

Loads component at file path.

**Parameters** **file_path** (*str*) – Location to load file.

**Returns** ComponentBase object

**needs_fitting**(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need to be fit or whose fit methods do nothing.

**Returns** True.

**property parameters**(*self*)

Returns the parameters which were used to initialize the component.

**predict**(*self*, *X: pandas.DataFrame*) → pandas.Series

Make predictions using selected features.

**Parameters** **X** (*pd.DataFrame*) – Data of shape [n_samples, n_features].

**Returns** Predicted values.

**Return type** pd.Series

**Raises** **MethodPropertyNotFoundError** – If estimator does not have a predict method or a component_obj that implements predict.

**predict_proba**(*self*, *X: pandas.DataFrame*) → pandas.Series

Make probability estimates for labels.

**Parameters** **X** (*pd.DataFrame*) – Features.

**Returns** Probability estimates.

**Return type** pd.Series

**Raises** **MethodPropertyNotFoundError** – If estimator does not have a predict_proba method or a component_obj that implements predict_proba.

**save**(*self*, *file_path*, *pickle_protocol=cloudpickle.DEFAULT_PROTOCOL*)

Saves component at file path.

**Parameters**

- **file_path** (*str*) – Location to save file.
- **pickle_protocol** (*int*) – The pickle data stream format.

**update_parameters**(*self*, *update_dict*, *reset_fit=True*)

Updates the parameter dictionary of the component.

**Parameters**

- **update_dict** (*dict*) – A dict of parameters to update.
- **reset_fit** (*bool, optional*) – If True, will set *_is_fitted* to False.

**class** evalml.pipelines.**ExtraTreesRegressor**(*n_estimators: int = 100*, *max_features: str = 'auto'*, *max_depth: int = 6*, *min_samples_split: int = 2*, *min_weight_fraction_leaf: float = 0.0*, *n_jobs: int = - 1*, *random_seed: Union[int, float] = 0*, ***kwargs*)

Extra Trees Regressor.

**Parameters**

- **n_estimators** (*float*) – The number of trees in the forest. Defaults to 100.
- **max_features** (*int, float or {"auto", "sqrt", "log2"}*) – The number of features to consider when looking for the best split:

- If int, then consider max_features features at each split.
- If float, then max_features is a fraction and int(max_features * n_features) features are considered at each split.
- If "auto", then max_features=sqrt(n_features).
- If "sqrt", then max_features=sqrt(n_features).
- If "log2", then max_features=log2(n_features).
- If None, then max_features = n_features.

The search for a split does not stop until at least one valid partition of the node samples is found, even if it requires to effectively inspect more than max_features features. Defaults to "auto".

- **max_depth** (*int*) – The maximum depth of the tree. Defaults to 6.
- **min_samples_split** (*int or float*) – The minimum number of samples required to split an internal node:
  - If int, then consider min_samples_split as the minimum number.
  - If float, then min_samples_split is a fraction and ceil(min_samples_split * n_samples) are the minimum number of samples for each split.
- **2.** (*Defaults to*) –
- **min_weight_fraction_leaf** (*float*) – The minimum weighted fraction of the sum total of weights (of all the input samples) required to be at a leaf node. Defaults to 0.0.
- **n_jobs** (*int or None*) – Number of jobs to run in parallel. -1 uses all processes. Defaults to -1.
- **random_seed** (*int*) – Seed for the random number generator. Defaults to 0.

**Attributes**

| | |
|---|---|
| **hyper-parameter_ranges** | { "n_estimators": Integer(10, 1000), "max_features": ["auto", "sqrt", "log2"], "max_depth": Integer(4, 10),} |
| **model_family** | ModelFamily.EXTRA_TREES |
| **modifies_features** | True |
| **modifies_target** | False |
| **name** | Extra Trees Regressor |
| **supported_problem_types** | [ ProblemTypes.REGRESSION, ProblemTypes.TIME_SERIES_REGRESSION,] |
| **training_only** | False |

**Methods**

| | |
|---|---|
| *clone* | Constructs a new component with the same parameters and random state. |
| *default_parameters* | Returns the default parameters for this component. |
| *describe* | Describe a component and its parameters. |
| *feature_importance* | Returns importance associated with each feature. |
| *fit* | Fits estimator to data. |
| *get_prediction_intervals* | Find the prediction intervals using the fitted ExtraTreesRegressor. |
| *load* | Loads component at file path. |
| *needs_fitting* | Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances. |
| *parameters* | Returns the parameters which were used to initialize the component. |
| *predict* | Make predictions using selected features. |
| *predict_proba* | Make probability estimates for labels. |
| *save* | Saves component at file path. |
| *update_parameters* | Updates the parameter dictionary of the component. |

**clone**(*self*)

Constructs a new component with the same parameters and random state.

**Returns** A new instance of this component with identical parameters and random state.

**default_parameters**(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

**Returns** Default parameters for this component.

**Return type** dict

**describe**(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

**Parameters**

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description as dictionary in the format {"name": name, "parameters": parameters}

**Returns** Returns dictionary if return_dict is True, else None.

**Return type** None or dict

**property feature_importance**(*self*) → pandas.Series

Returns importance associated with each feature.

**Returns** Importance associated with each feature.

**Return type** np.ndarray

**Raises** **MethodPropertyNotFoundError** – If estimator does not have a feature_importance method or a component_obj that implements feature_importance.

**fit**(*self*, *X: pandas.DataFrame*, *y: Optional[pandas.Series] = None*)

Fits estimator to data.

**Parameters**

- **X** (*pd.DataFrame*) – The input training data of shape [n_samples, n_features].
- **y** (*pd.Series, optional*) – The target training data of length [n_samples].

**Returns** self

**get_prediction_intervals**(*self, X: pandas.DataFrame, y: Optional[pandas.Series] = None, coverage: List[float] = None, predictions: pandas.Series = None*) → Dict[str, pandas.Series]

Find the prediction intervals using the fitted ExtraTreesRegressor.

**Parameters**

- **X** (*pd.DataFrame*) – Data of shape [n_samples, n_features].
- **y** (*pd.Series*) – Target data. Optional.
- **coverage** (*list[float]*) – A list of floats between the values 0 and 1 that the upper and lower bounds of the prediction interval should be calculated for.
- **predictions** (*pd.Series*) – Optional list of predictions to use. If None, will generate predictions using *X*.

**Returns** Prediction intervals, keys are in the format {coverage}_lower or {coverage}_upper.

**Return type** dict

**static load**(*file_path*)

Loads component at file path.

**Parameters** **file_path** (*str*) – Location to load file.

**Returns** ComponentBase object

**needs_fitting**(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need to be fit or whose fit methods do nothing.

**Returns** True.

**property parameters**(*self*)

Returns the parameters which were used to initialize the component.

**predict**(*self, X: pandas.DataFrame*) → pandas.Series

Make predictions using selected features.

**Parameters** **X** (*pd.DataFrame*) – Data of shape [n_samples, n_features].

**Returns** Predicted values.

**Return type** pd.Series

**Raises** **MethodPropertyNotFoundError** – If estimator does not have a predict method or a component_obj that implements predict.

**predict_proba**(*self, X: pandas.DataFrame*) → pandas.Series

Make probability estimates for labels.

**Parameters** **X** (*pd.DataFrame*) – Features.

**Returns** Probability estimates.

**Return type** pd.Series

**Raises** `MethodPropertyNotFoundError` – If estimator does not have a predict_proba method or a component_obj that implements predict_proba.

`save`(*self*, *file_path*, *pickle_protocol=cloudpickle.DEFAULT_PROTOCOL*)

Saves component at file path.

**Parameters**

- `file_path` (*str*) – Location to save file.
- `pickle_protocol` (*int*) – The pickle data stream format.

`update_parameters`(*self*, *update_dict*, *reset_fit=True*)

Updates the parameter dictionary of the component.

**Parameters**

- `update_dict` (*dict*) – A dict of parameters to update.
- `reset_fit` (*bool, optional*) – If True, will set *_is_fitted* to False.

`class` `evalml.pipelines.`**`FeatureSelector`**(*parameters=None*, *component_obj=None*, *random_seed=0*, ***kwargs*)

Selects top features based on importance weights.

**Parameters**

- `parameters` (*dict*) – Dictionary of parameters for the component. Defaults to None.
- `component_obj` (*obj*) – Third-party objects useful in component implementation. Defaults to None.
- `random_seed` (*int*) – Seed for the random number generator. Defaults to 0.

**Attributes**

| | |
|---|---|
| **modifies_features** | True |
| **modifies_target** | False |
| **training_only** | False |

**Methods**

| | |
|---|---|
| *clone* | Constructs a new component with the same parameters and random state. |
| *default_parameters* | Returns the default parameters for this component. |
| *describe* | Describe a component and its parameters. |
| *fit* | Fits component to data. |
| *fit_transform* | Fit and transform data using the feature selector. |
| *get_names* | Get names of selected features. |
| *load* | Loads component at file path. |
| *name* | Returns string name of this component. |
| *needs_fitting* | Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances. |
| *parameters* | Returns the parameters which were used to initialize the component. |
| *save* | Saves component at file path. |
| *transform* | Transforms input data by selecting features. If the component_obj does not have a transform method, will raise an MethodPropertyNotFoundError exception. |
| *update_parameters* | Updates the parameter dictionary of the component. |

**clone**(*self*)

Constructs a new component with the same parameters and random state.

**Returns** A new instance of this component with identical parameters and random state.

**default_parameters**(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

**Returns** Default parameters for this component.

**Return type** dict

**describe**(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

**Parameters**

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description as dictionary in the format {"name": name, "parameters": parameters}

**Returns** Returns dictionary if return_dict is True, else None.

**Return type** None or dict

**fit**(*self*, *X*, *y=None*)

Fits component to data.

**Parameters**

- **X** (*pd.DataFrame*) – The input training data of shape [n_samples, n_features]
- **y** (*pd.Series, optional*) – The target training data of length [n_samples]

**Returns** self

**Raises** `MethodPropertyNotFoundError` – If component does not have a fit method or a component_obj that implements fit.

`fit_transform`(*self*, *X*, *y=None*)

Fit and transform data using the feature selector.

**Parameters**

- `X` (*pd.DataFrame*) – The input training data of shape [n_samples, n_features].
- `y` (*pd.Series, optional*) – The target training data of length [n_samples].

**Returns** Transformed data.

**Return type** pd.DataFrame

`get_names`(*self*)

Get names of selected features.

**Returns** List of the names of features selected.

**Return type** list[str]

`static load`(*file_path*)

Loads component at file path.

**Parameters** `file_path` (*str*) – Location to load file.

**Returns** ComponentBase object

`property name`(*cls*)

Returns string name of this component.

`needs_fitting`(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need to be fit or whose fit methods do nothing.

**Returns** True.

`property parameters`(*self*)

Returns the parameters which were used to initialize the component.

`save`(*self*, *file_path*, *pickle_protocol=cloudpickle.DEFAULT_PROTOCOL*)

Saves component at file path.

**Parameters**

- `file_path` (*str*) – Location to save file.
- `pickle_protocol` (*int*) – The pickle data stream format.

`transform`(*self*, *X*, *y=None*)

Transforms input data by selecting features. If the component_obj does not have a transform method, will raise an MethodPropertyNotFoundError exception.

**Parameters**

- `X` (*pd.DataFrame*) – Data to transform.
- `y` (*pd.Series, optional*) – Target data. Ignored.

**Returns** Transformed X

**Return type** pd.DataFrame

**Raises** **MethodPropertyNotFoundError** – If feature selector does not have a transform method or a component_obj that implements transform

**update_parameters**(*self*, *update_dict*, *reset_fit=True*)

Updates the parameter dictionary of the component.

**Parameters**

- **update_dict** (*dict*) – A dict of parameters to update.
- **reset_fit** (*bool, optional*) – If True, will set *_is_fitted* to False.

**class** evalml.pipelines.**Imputer**(*categorical_impute_strategy='most_frequent'*, *categorical_fill_value=None*, *numeric_impute_strategy='mean'*, *numeric_fill_value=None*, *boolean_impute_strategy='most_frequent'*, *boolean_fill_value=None*, *random_seed=0*, ***kwargs*)

Imputes missing data according to a specified imputation strategy.

**Parameters**

- **categorical_impute_strategy** (*string*) – Impute strategy to use for string, object, boolean, categorical dtypes. Valid values include "most_frequent" and "constant".
- **numeric_impute_strategy** (*string*) – Impute strategy to use for numeric columns. Valid values include "mean", "median", "most_frequent", and "constant".
- **boolean_impute_strategy** (*string*) – Impute strategy to use for boolean columns. Valid values include "most_frequent" and "constant".
- **categorical_fill_value** (*string*) – When categorical_impute_strategy == "constant", fill_value is used to replace missing data. The default value of None will fill with the string "missing_value".
- **numeric_fill_value** (*int, float*) – When numeric_impute_strategy == "constant", fill_value is used to replace missing data. The default value of None will fill with 0.
- **boolean_fill_value** (*bool*) – When boolean_impute_strategy == "constant", fill_value is used to replace missing data. The default value of None will fill with True.
- **random_seed** (*int*) – Seed for the random number generator. Defaults to 0.

**Attributes**

| | |
|---|---|
| **hyper-parameter_ranges** | { "categorical_impute_strategy": ["most_frequent"], "numeric_impute_strategy": ["mean", "median", "most_frequent", "knn"], "boolean_impute_strategy": ["most_frequent"]} |
| **modifies_features** | True |
| **modifies_target** | False |
| **name** | Imputer |
| **training_only** | False |

**Methods**

| | |
|---|---|
| *clone* | Constructs a new component with the same parameters and random state. |
| *default_parameters* | Returns the default parameters for this component. |
| *describe* | Describe a component and its parameters. |
| *fit* | Fits imputer to data. 'None' values are converted to np.nan before imputation and are treated as the same. |
| *fit_transform* | Fits on X and transforms X. |
| *load* | Loads component at file path. |
| *needs_fitting* | Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances. |
| *parameters* | Returns the parameters which were used to initialize the component. |
| *save* | Saves component at file path. |
| *transform* | Transforms data X by imputing missing values. |
| *update_parameters* | Updates the parameter dictionary of the component. |

**clone**(*self*)

Constructs a new component with the same parameters and random state.

**Returns** A new instance of this component with identical parameters and random state.

**default_parameters**(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

**Returns** Default parameters for this component.

**Return type** dict

**describe**(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

**Parameters**

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description as dictionary in the format {"name": name, "parameters": parameters}

**Returns** Returns dictionary if return_dict is True, else None.

**Return type** None or dict

**fit**(*self*, *X*, *y=None*)

Fits imputer to data. 'None' values are converted to np.nan before imputation and are treated as the same.

**Parameters**

- **X** (*pd.DataFrame, np.ndarray*) – The input training data of shape [n_samples, n_features]
- **y** (*pd.Series, optional*) – The target training data of length [n_samples]

**Returns** self

**fit_transform**(*self*, *X*, *y=None*)

Fits on X and transforms X.

**Parameters**

- **X** (*pd.DataFrame*) – Data to fit and transform.
- **y** (*pd.Series*) – Target data.

**Returns** Transformed X.

**Return type** pd.DataFrame

**Raises** **MethodPropertyNotFoundError** – If transformer does not have a transform method or a component_obj that implements transform.

**static load**(*file_path*)

Loads component at file path.

**Parameters** **file_path** (*str*) – Location to load file.

**Returns** ComponentBase object

**needs_fitting**(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need to be fit or whose fit methods do nothing.

**Returns** True.

**property parameters**(*self*)

Returns the parameters which were used to initialize the component.

**save**(*self*, *file_path*, *pickle_protocol=cloudpickle.DEFAULT_PROTOCOL*)

Saves component at file path.

**Parameters**

- **file_path** (*str*) – Location to save file.
- **pickle_protocol** (*int*) – The pickle data stream format.

**transform**(*self*, *X*, *y=None*)

Transforms data X by imputing missing values.

**Parameters**

- **X** (*pd.DataFrame*) – Data to transform
- **y** (*pd.Series, optional*) – Ignored.

**Returns** Transformed X

**Return type** pd.DataFrame

**update_parameters**(*self*, *update_dict*, *reset_fit=True*)

Updates the parameter dictionary of the component.

**Parameters**

- **update_dict** (*dict*) – A dict of parameters to update.
- **reset_fit** (*bool, optional*) – If True, will set *_is_fitted* to False.

**class** evalml.pipelines.**KNeighborsClassifier**(*n_neighbors=5*, *weights='uniform'*, *algorithm='auto'*, *leaf_size=30*, *p=2*, *random_seed=0*, ***kwargs*)

K-Nearest Neighbors Classifier.

**Parameters**

- **n_neighbors** (*int*) – Number of neighbors to use by default. Defaults to 5.
- **weights** (*{'uniform', 'distance'} or callable*) – Weight function used in prediction. Can be:
  - 'uniform' : uniform weights. All points in each neighborhood are weighted equally.
  - 'distance' : weight points by the inverse of their distance. in this case, closer neighbors of a query point will have a greater influence than neighbors which are further away.
  - [callable] : a user-defined function which accepts an array of distances, and returns an array of the same shape containing the weights.

  Defaults to "uniform".
- **algorithm** (*{'auto', 'ball_tree', 'kd_tree', 'brute'}*) – Algorithm used to compute the nearest neighbors:
  - 'ball_tree' will use BallTree
  - 'kd_tree' will use KDTree
  - 'brute' will use a brute-force search.

  'auto' will attempt to decide the most appropriate algorithm based on the values passed to fit method. Defaults to "auto". Note: fitting on sparse input will override the setting of this parameter, using brute force.
- **leaf_size** (*int*) – Leaf size passed to BallTree or KDTree. This can affect the speed of the construction and query, as well as the memory required to store the tree. The optimal value depends on the nature of the problem. Defaults to 30.
- **p** (*int*) – Power parameter for the Minkowski metric. When p = 1, this is equivalent to using manhattan_distance (l1), and euclidean_distance (l2) for p = 2. For arbitrary p, minkowski_distance (l_p) is used. Defaults to 2.
- **random_seed** (*int*) – Seed for the random number generator. Defaults to 0.

**Attributes**

| | |
|---|---|
| **hyper-parameter_ranges** | { "n_neighbors": Integer(2, 12), "weights": ["uniform", "distance"], "algorithm": ["auto", "ball_tree", "kd_tree", "brute"], "leaf_size": Integer(10, 30), "p": Integer(1, 5),} |
| **model_family** | ModelFamily.K_NEIGHBORS |
| **modifies_features** | True |
| **modifies_target** | False |
| **name** | KNN Classifier |
| **supported_problem_types** | [ ProblemTypes.BINARY, ProblemTypes.MULTICLASS, ProblemTypes.TIME_SERIES_BINARY, ProblemTypes.TIME_SERIES_MULTICLASS,] |
| **training_only** | False |

**Methods**

| | |
|---|---|
| *clone* | Constructs a new component with the same parameters and random state. |
| *default_parameters* | Returns the default parameters for this component. |
| *describe* | Describe a component and its parameters. |
| *feature_importance* | Returns array of 0's matching the input number of features as feature_importance is not defined for KNN classifiers. |
| *fit* | Fits estimator to data. |
| *get_prediction_intervals* | Find the prediction intervals using the fitted regressor. |
| *load* | Loads component at file path. |
| *needs_fitting* | Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances. |
| *parameters* | Returns the parameters which were used to initialize the component. |
| *predict* | Make predictions using selected features. |
| *predict_proba* | Make probability estimates for labels. |
| *save* | Saves component at file path. |
| *update_parameters* | Updates the parameter dictionary of the component. |

**clone**(*self*)

Constructs a new component with the same parameters and random state.

**Returns** A new instance of this component with identical parameters and random state.

**default_parameters**(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

**Returns** Default parameters for this component.

**Return type** dict

**describe**(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

**Parameters**

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description as dictionary in the format {"name": name, "parameters": parameters}

**Returns** Returns dictionary if return_dict is True, else None.

**Return type** None or dict

**property feature_importance**(*self*)

Returns array of 0's matching the input number of features as feature_importance is not defined for KNN classifiers.

**fit**(*self*, *X: pandas.DataFrame*, *y: Optional[pandas.Series] = None*)

Fits estimator to data.

**Parameters**

- **X** (*pd.DataFrame*) – The input training data of shape [n_samples, n_features].

- **y** (*pd.Series, optional*) – The target training data of length [n_samples].

**Returns** self

**get_prediction_intervals**(*self, X: pandas.DataFrame, y: Optional[pandas.Series] = None, coverage: List[float] = None, predictions: pandas.Series = None*) → Dict[str, pandas.Series]

Find the prediction intervals using the fitted regressor.

This function takes the predictions of the fitted estimator and calculates the rolling standard deviation across all predictions using a window size of 5. The lower and upper predictions are determined by taking the percent point (quantile) function of the lower tail probability at each bound multiplied by the rolling standard deviation.

**Parameters**

- **X** (*pd.DataFrame*) – Data of shape [n_samples, n_features].
- **y** (*pd.Series*) – Target data. Ignored.
- **coverage** (*list[float]*) – A list of floats between the values 0 and 1 that the upper and lower bounds of the prediction interval should be calculated for.
- **predictions** (*pd.Series*) – Optional list of predictions to use. If None, will generate predictions using *X*.

**Returns** Prediction intervals, keys are in the format {coverage}_lower or {coverage}_upper.

**Return type** dict

**Raises** **MethodPropertyNotFoundError** – If the estimator does not support Time Series Regression as a problem type.

**static load**(*file_path*)

Loads component at file path.

**Parameters** **file_path** (*str*) – Location to load file.

**Returns** ComponentBase object

**needs_fitting**(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need to be fit or whose fit methods do nothing.

**Returns** True.

**property parameters**(*self*)

Returns the parameters which were used to initialize the component.

**predict**(*self, X: pandas.DataFrame*) → pandas.Series

Make predictions using selected features.

**Parameters** **X** (*pd.DataFrame*) – Data of shape [n_samples, n_features].

**Returns** Predicted values.

**Return type** pd.Series

**Raises** **MethodPropertyNotFoundError** – If estimator does not have a predict method or a component_obj that implements predict.

**predict_proba**(*self*, *X: pandas.DataFrame*) → pandas.Series

Make probability estimates for labels.

**Parameters** **X** (*pd.DataFrame*) – Features.

**Returns** Probability estimates.

**Return type** pd.Series

**Raises** **MethodPropertyNotFoundError** – If estimator does not have a predict_proba method or a component_obj that implements predict_proba.

**save**(*self*, *file_path*, *pickle_protocol=cloudpickle.DEFAULT_PROTOCOL*)

Saves component at file path.

**Parameters**

- **file_path** (*str*) – Location to save file.
- **pickle_protocol** (*int*) – The pickle data stream format.

**update_parameters**(*self*, *update_dict*, *reset_fit=True*)

Updates the parameter dictionary of the component.

**Parameters**

- **update_dict** (*dict*) – A dict of parameters to update.
- **reset_fit** (*bool, optional*) – If True, will set *_is_fitted* to False.

class evalml.pipelines.**LightGBMClassifier**(*boosting_type='gbdt'*, *learning_rate=0.1*, *n_estimators=100*, *max_depth=0*, *num_leaves=31*, *min_child_samples=20*, *bagging_fraction=0.9*, *bagging_freq=0*, *n_jobs=- 1*, *random_seed=0*, ***kwargs*)

LightGBM Classifier.

**Parameters**

- **boosting_type** (*string*) – Type of boosting to use. Defaults to "gbdt". - 'gbdt' uses traditional Gradient Boosting Decision Tree - "dart", uses Dropouts meet Multiple Additive Regression Trees - "goss", uses Gradient-based One-Side Sampling - "rf", uses Random Forest
- **learning_rate** (*float*) – Boosting learning rate. Defaults to 0.1.
- **n_estimators** (*int*) – Number of boosted trees to fit. Defaults to 100.
- **max_depth** (*int*) – Maximum tree depth for base learners, <=0 means no limit. Defaults to 0.
- **num_leaves** (*int*) – Maximum tree leaves for base learners. Defaults to 31.
- **min_child_samples** (*int*) – Minimum number of data needed in a child (leaf). Defaults to 20.
- **bagging_fraction** (*float*) – LightGBM will randomly select a subset of features on each iteration (tree) without resampling if this is smaller than 1.0. For example, if set to 0.8, LightGBM will select 80% of features before training each tree. This can be used to speed up training and deal with overfitting. Defaults to 0.9.
- **bagging_freq** (*int*) – Frequency for bagging. 0 means bagging is disabled. k means perform bagging at every k iteration. Every k-th iteration, LightGBM will randomly select bagging_fraction * 100 % of the data to use for the next k iterations. Defaults to 0.

- **n_jobs** (*int or None*) – Number of threads to run in parallel. -1 uses all threads. Defaults to -1.
- **random_seed** (*int*) – Seed for the random number generator. Defaults to 0.

**Attributes**

| | |
|---|---|
| **hyper-parameter_ranges** | { "learning_rate": Real(0.000001, 1), "boosting_type": ["gbdt", "dart", "goss", "rf"], "n_estimators": Integer(10, 100), "max_depth": Integer(0, 10), "num_leaves": Integer(2, 100), "min_child_samples": Integer(1, 100), "bagging_fraction": Real(0.000001, 1), "bagging_freq": Integer(0, 1),} |
| **model_family** | ModelFamily.LIGHTGBM |
| **modifies_features** | True |
| **modifies_target** | False |
| **name** | LightGBM Classifier |
| **SEED_MAX** | SEED_BOUNDS.max_bound |
| **SEED_MIN** | 0 |
| **supported_problem_types** | [ ProblemTypes.BINARY, ProblemTypes.MULTICLASS, ProblemTypes.TIME_SERIES_BINARY, ProblemTypes.TIME_SERIES_MULTICLASS,] |
| **training_only** | False |

**Methods**

| | |
|---|---|
| *clone* | Constructs a new component with the same parameters and random state. |
| *default_parameters* | Returns the default parameters for this component. |
| *describe* | Describe a component and its parameters. |
| *feature_importance* | Returns importance associated with each feature. |
| *fit* | Fits LightGBM classifier component to data. |
| *get_prediction_intervals* | Find the prediction intervals using the fitted regressor. |
| *load* | Loads component at file path. |
| *needs_fitting* | Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances. |
| *parameters* | Returns the parameters which were used to initialize the component. |
| *predict* | Make predictions using the fitted LightGBM classifier. |
| *predict_proba* | Make prediction probabilities using the fitted LightGBM classifier. |
| *save* | Saves component at file path. |
| *update_parameters* | Updates the parameter dictionary of the component. |

**clone**(*self*)

Constructs a new component with the same parameters and random state.

**Returns** A new instance of this component with identical parameters and random state.

**default_parameters**(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

**Returns** Default parameters for this component.

**Return type** dict

**describe**(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

**Parameters**

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description as dictionary in the format {"name": name, "parameters": parameters}

**Returns** Returns dictionary if return_dict is True, else None.

**Return type** None or dict

**property feature_importance**(*self*) → pandas.Series

Returns importance associated with each feature.

**Returns** Importance associated with each feature.

**Return type** np.ndarray

**Raises** **MethodPropertyNotFoundError** – If estimator does not have a feature_importance method or a component_obj that implements feature_importance.

**fit**(*self*, *X*, *y=None*)

Fits LightGBM classifier component to data.

**Parameters**

- **X** (*pd.DataFrame*) – The input training data of shape [n_samples, n_features].
- **y** (*pd.Series*) – The target training data of length [n_samples].

**Returns** self

**get_prediction_intervals**(*self*, *X: pandas.DataFrame*, *y: Optional[pandas.Series] = None*, *coverage: List[float] = None*, *predictions: pandas.Series = None*) → Dict[str, pandas.Series]

Find the prediction intervals using the fitted regressor.

This function takes the predictions of the fitted estimator and calculates the rolling standard deviation across all predictions using a window size of 5. The lower and upper predictions are determined by taking the percent point (quantile) function of the lower tail probability at each bound multiplied by the rolling standard deviation.

**Parameters**

- **X** (*pd.DataFrame*) – Data of shape [n_samples, n_features].
- **y** (*pd.Series*) – Target data. Ignored.
- **coverage** (*list[float]*) – A list of floats between the values 0 and 1 that the upper and lower bounds of the prediction interval should be calculated for.
- **predictions** (*pd.Series*) – Optional list of predictions to use. If None, will generate predictions using *X*.

**Returns** Prediction intervals, keys are in the format {coverage}_lower or {coverage}_upper.

**Return type** dict

**Raises** `MethodPropertyNotFoundError` – If the estimator does not support Time Series Regression as a problem type.

`static load`(*file_path*)

Loads component at file path.

**Parameters** `file_path` (*str*) – Location to load file.

**Returns** ComponentBase object

`needs_fitting`(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need to be fit or whose fit methods do nothing.

**Returns** True.

`property parameters`(*self*)

Returns the parameters which were used to initialize the component.

`predict`(*self*, *X*)

Make predictions using the fitted LightGBM classifier.

**Parameters** `X` (*pd.DataFrame*) – Data of shape [n_samples, n_features].

**Returns** Predicted values.

**Return type** pd.DataFrame

`predict_proba`(*self*, *X*)

Make prediction probabilities using the fitted LightGBM classifier.

**Parameters** `X` (*pd.DataFrame*) – Data of shape [n_samples, n_features].

**Returns** Predicted probability values.

**Return type** pd.DataFrame

`save`(*self*, *file_path*, *pickle_protocol=cloudpickle.DEFAULT_PROTOCOL*)

Saves component at file path.

**Parameters**

- `file_path` (*str*) – Location to save file.
- `pickle_protocol` (*int*) – The pickle data stream format.

`update_parameters`(*self*, *update_dict*, *reset_fit=True*)

Updates the parameter dictionary of the component.

**Parameters**

- `update_dict` (*dict*) – A dict of parameters to update.
- `reset_fit` (*bool, optional*) – If True, will set *_is_fitted* to False.

`class` evalml.pipelines.`LightGBMRegressor`(*boosting_type='gbdt', learning_rate=0.1, n_estimators=20, max_depth=0, num_leaves=31, min_child_samples=20, bagging_fraction=0.9, bagging_freq=0, n_jobs=- 1, random_seed=0, **kwargs*)

LightGBM Regressor.

**Parameters**

- **boosting_type** (*string*) – Type of boosting to use. Defaults to "gbdt". - 'gbdt' uses traditional Gradient Boosting Decision Tree - "dart", uses Dropouts meet Multiple Additive Regression Trees - "goss", uses Gradient-based One-Side Sampling - "rf", uses Random Forest
- **learning_rate** (*float*) – Boosting learning rate. Defaults to 0.1.
- **n_estimators** (*int*) – Number of boosted trees to fit. Defaults to 100.
- **max_depth** (*int*) – Maximum tree depth for base learners, <=0 means no limit. Defaults to 0.
- **num_leaves** (*int*) – Maximum tree leaves for base learners. Defaults to 31.
- **min_child_samples** (*int*) – Minimum number of data needed in a child (leaf). Defaults to 20.
- **bagging_fraction** (*float*) – LightGBM will randomly select a subset of features on each iteration (tree) without resampling if this is smaller than 1.0. For example, if set to 0.8, LightGBM will select 80% of features before training each tree. This can be used to speed up training and deal with overfitting. Defaults to 0.9.
- **bagging_freq** (*int*) – Frequency for bagging. 0 means bagging is disabled. k means perform bagging at every k iteration. Every k-th iteration, LightGBM will randomly select bagging_fraction * 100 % of the data to use for the next k iterations. Defaults to 0.
- **n_jobs** (*int or None*) – Number of threads to run in parallel. -1 uses all threads. Defaults to -1.
- **random_seed** (*int*) – Seed for the random number generator. Defaults to 0.

**Attributes**

| | |
|---|---|
| **hyper-parameter_ranges** | { "learning_rate": Real(0.000001, 1), "boosting_type": ["gbdt", "dart", "goss", "rf"], "n_estimators": Integer(10, 100), "max_depth": Integer(0, 10), "num_leaves": Integer(2, 100), "min_child_samples": Integer(1, 100), "bagging_fraction": Real(0.000001, 1), "bagging_freq": Integer(0, 1),} |
| **model_family** | ModelFamily.LIGHTGBM |
| **modifies_features** | True |
| **modifies_target** | False |
| **name** | LightGBM Regressor |
| **SEED_MAX** | SEED_BOUNDS.max_bound |
| **SEED_MIN** | 0 |
| **supported_problem_types** | [ProblemTypes.REGRESSION] |
| **training_only** | False |

**Methods**

| | |
|---|---|
| *clone* | Constructs a new component with the same parameters and random state. |
| *default_parameters* | Returns the default parameters for this component. |
| *describe* | Describe a component and its parameters. |
| *feature_importance* | Returns importance associated with each feature. |
| *fit* | Fits LightGBM regressor to data. |
| *get_prediction_intervals* | Find the prediction intervals using the fitted regressor. |
| *load* | Loads component at file path. |
| *needs_fitting* | Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances. |
| *parameters* | Returns the parameters which were used to initialize the component. |
| *predict* | Make predictions using fitted LightGBM regressor. |
| *predict_proba* | Make probability estimates for labels. |
| *save* | Saves component at file path. |
| *update_parameters* | Updates the parameter dictionary of the component. |

**clone**(*self*)

Constructs a new component with the same parameters and random state.

**Returns** A new instance of this component with identical parameters and random state.

**default_parameters**(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

**Returns** Default parameters for this component.

**Return type** dict

**describe**(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

**Parameters**

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description as dictionary in the format {"name": name, "parameters": parameters}

**Returns** Returns dictionary if return_dict is True, else None.

**Return type** None or dict

**property feature_importance**(*self*) → pandas.Series

Returns importance associated with each feature.

**Returns** Importance associated with each feature.

**Return type** np.ndarray

**Raises** **MethodPropertyNotFoundError** – If estimator does not have a feature_importance method or a component_obj that implements feature_importance.

**fit**(*self*, *X*, *y=None*)

Fits LightGBM regressor to data.

**Parameters**

- **X** (*pd.DataFrame*) – The input training data of shape [n_samples, n_features].
- **y** (*pd.Series*) – The target training data of length [n_samples].

**Returns** self

**get_prediction_intervals**(*self, X: pandas.DataFrame, y: Optional[pandas.Series] = None, coverage: List[float] = None, predictions: pandas.Series = None*) → Dict[str, pandas.Series]

Find the prediction intervals using the fitted regressor.

This function takes the predictions of the fitted estimator and calculates the rolling standard deviation across all predictions using a window size of 5. The lower and upper predictions are determined by taking the percent point (quantile) function of the lower tail probability at each bound multiplied by the rolling standard deviation.

**Parameters**

- **X** (*pd.DataFrame*) – Data of shape [n_samples, n_features].
- **y** (*pd.Series*) – Target data. Ignored.
- **coverage** (*list[float]*) – A list of floats between the values 0 and 1 that the upper and lower bounds of the prediction interval should be calculated for.
- **predictions** (*pd.Series*) – Optional list of predictions to use. If None, will generate predictions using *X*.

**Returns** Prediction intervals, keys are in the format {coverage}_lower or {coverage}_upper.

**Return type** dict

**Raises** **MethodPropertyNotFoundError** – If the estimator does not support Time Series Regression as a problem type.

**static load**(*file_path*)

Loads component at file path.

**Parameters** **file_path** (*str*) – Location to load file.

**Returns** ComponentBase object

**needs_fitting**(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need to be fit or whose fit methods do nothing.

**Returns** True.

**property parameters**(*self*)

Returns the parameters which were used to initialize the component.

**predict**(*self, X*)

Make predictions using fitted LightGBM regressor.

**Parameters** **X** (*pd.DataFrame*) – Data of shape [n_samples, n_features].

**Returns** Predicted values.

**Return type** pd.Series

**predict_proba**(*self*, *X: pandas.DataFrame*) → pandas.Series

Make probability estimates for labels.

**Parameters** **X** (*pd.DataFrame*) – Features.

**Returns** Probability estimates.

**Return type** pd.Series

**Raises** **MethodPropertyNotFoundError** – If estimator does not have a predict_proba method or a component_obj that implements predict_proba.

**save**(*self*, *file_path*, *pickle_protocol=cloudpickle.DEFAULT_PROTOCOL*)

Saves component at file path.

**Parameters**

- **file_path** (*str*) – Location to save file.
- **pickle_protocol** (*int*) – The pickle data stream format.

**update_parameters**(*self*, *update_dict*, *reset_fit=True*)

Updates the parameter dictionary of the component.

**Parameters**

- **update_dict** (*dict*) – A dict of parameters to update.
- **reset_fit** (*bool, optional*) – If True, will set *_is_fitted* to False.

**class** evalml.pipelines.**LinearRegressor**(*fit_intercept=True*, *n_jobs=- 1*, *random_seed=0*, ***kwargs*)

Linear Regressor.

**Parameters**

- **fit_intercept** (*boolean*) – Whether to calculate the intercept for this model. If set to False, no intercept will be used in calculations (i.e. data is expected to be centered). Defaults to True.
- **n_jobs** (*int or None*) – Number of jobs to run in parallel. -1 uses all threads. Defaults to -1.
- **random_seed** (*int*) – Seed for the random number generator. Defaults to 0.

**Attributes**

| | |
|---|---|
| **hyper-parameter_ranges** | { "fit_intercept": [True, False],} |
| **model_family** | ModelFamily.LINEAR_MODEL |
| **modifies_features** | True |
| **modifies_target** | False |
| **name** | Linear Regressor |
| **supported_problem_types** | [ ProblemTypes.REGRESSION, ProblemTypes.TIME_SERIES_REGRESSION,] |
| **training_only** | False |

**Methods**

| | |
|---|---|
| *clone* | Constructs a new component with the same parameters and random state. |
| *default_parameters* | Returns the default parameters for this component. |
| *describe* | Describe a component and its parameters. |
| *feature_importance* | Feature importance for fitted linear regressor. |
| *fit* | Fits estimator to data. |
| *get_prediction_intervals* | Find the prediction intervals using the fitted regressor. |
| *load* | Loads component at file path. |
| *needs_fitting* | Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances. |
| *parameters* | Returns the parameters which were used to initialize the component. |
| *predict* | Make predictions using selected features. |
| *predict_proba* | Make probability estimates for labels. |
| *save* | Saves component at file path. |
| *update_parameters* | Updates the parameter dictionary of the component. |

**clone**(*self*)

Constructs a new component with the same parameters and random state.

**Returns** A new instance of this component with identical parameters and random state.

**default_parameters**(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

**Returns** Default parameters for this component.

**Return type** dict

**describe**(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

**Parameters**

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description as dictionary in the format {"name": name, "parameters": parameters}

**Returns** Returns dictionary if return_dict is True, else None.

**Return type** None or dict

**property feature_importance**(*self*)

Feature importance for fitted linear regressor.

**fit**(*self*, *X: pandas.DataFrame*, *y: Optional[pandas.Series] = None*)

Fits estimator to data.

**Parameters**

- **X** (*pd.DataFrame*) – The input training data of shape [n_samples, n_features].
- **y** (*pd.Series, optional*) – The target training data of length [n_samples].

**Returns** self

**get_prediction_intervals**(*self*, *X: pandas.DataFrame*, *y: Optional[pandas.Series] = None*, *coverage: List[float] = None*, *predictions: pandas.Series = None*) → Dict[str, pandas.Series]

Find the prediction intervals using the fitted regressor.

This function takes the predictions of the fitted estimator and calculates the rolling standard deviation across all predictions using a window size of 5. The lower and upper predictions are determined by taking the percent point (quantile) function of the lower tail probability at each bound multiplied by the rolling standard deviation.

**Parameters**

- **X** (*pd.DataFrame*) – Data of shape [n_samples, n_features].
- **y** (*pd.Series*) – Target data. Ignored.
- **coverage** (*list[float]*) – A list of floats between the values 0 and 1 that the upper and lower bounds of the prediction interval should be calculated for.
- **predictions** (*pd.Series*) – Optional list of predictions to use. If None, will generate predictions using *X*.

**Returns** Prediction intervals, keys are in the format {coverage}_lower or {coverage}_upper.

**Return type** dict

**Raises** **MethodPropertyNotFoundError** – If the estimator does not support Time Series Regression as a problem type.

**static load**(*file_path*)

Loads component at file path.

**Parameters** **file_path** (*str*) – Location to load file.

**Returns** ComponentBase object

**needs_fitting**(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need to be fit or whose fit methods do nothing.

**Returns** True.

**property parameters**(*self*)

Returns the parameters which were used to initialize the component.

**predict**(*self*, *X: pandas.DataFrame*) → pandas.Series

Make predictions using selected features.

**Parameters** **X** (*pd.DataFrame*) – Data of shape [n_samples, n_features].

**Returns** Predicted values.

**Return type** pd.Series

**Raises** **MethodPropertyNotFoundError** – If estimator does not have a predict method or a component_obj that implements predict.

**predict_proba**(*self*, *X: pandas.DataFrame*) → pandas.Series

Make probability estimates for labels.

**Parameters** **X** (*pd.DataFrame*) – Features.

**Returns** Probability estimates.

**Return type** pd.Series

**Raises** `MethodPropertyNotFoundError` – If estimator does not have a predict_proba method or a component_obj that implements predict_proba.

`save`(*self*, *file_path*, *pickle_protocol=cloudpickle.DEFAULT_PROTOCOL*)

Saves component at file path.

**Parameters**

- `file_path` (*str*) – Location to save file.
- `pickle_protocol` (*int*) – The pickle data stream format.

`update_parameters`(*self*, *update_dict*, *reset_fit=True*)

Updates the parameter dictionary of the component.

**Parameters**

- `update_dict` (*dict*) – A dict of parameters to update.
- `reset_fit` (*bool, optional*) – If True, will set *_is_fitted* to False.

**class** evalml.pipelines.`LogisticRegressionClassifier`(*penalty='l2'*, *C=1.0*, *multi_class='auto'*, *solver='lbfgs'*, *n_jobs=- 1*, *random_seed=0*, ***kwargs*)

Logistic Regression Classifier.

**Parameters**

- `penalty` (*{"l1", "l2", "elasticnet", "none"}*) – The norm used in penalization. Defaults to "l2".
- `C` (*float*) – Inverse of regularization strength. Must be a positive float. Defaults to 1.0.
- `multi_class` (*{"auto", "ovr", "multinomial"}*) – If the option chosen is "ovr", then a binary problem is fit for each label. For "multinomial" the loss minimised is the multinomial loss fit across the entire probability distribution, even when the data is binary. "multinomial" is unavailable when solver="liblinear". "auto" selects "ovr" if the data is binary, or if solver="liblinear", and otherwise selects "multinomial". Defaults to "auto".
- `solver` (*{"newton-cg", "lbfgs", "liblinear", "sag", "saga"}*) – Algorithm to use in the optimization problem. For small datasets, "liblinear" is a good choice, whereas "sag" and "saga" are faster for large ones. For multiclass problems, only "newton-cg", "sag", "saga" and "lbfgs" handle multinomial loss; "liblinear" is limited to one-versus-rest schemes.
  - "newton-cg", "lbfgs", "sag" and "saga" handle L2 or no penalty
  - "liblinear" and "saga" also handle L1 penalty
  - "saga" also supports "elasticnet" penalty
  - "liblinear" does not support setting penalty='none'

  Defaults to "lbfgs".
- `n_jobs` (*int*) – Number of parallel threads used to run xgboost. Note that creating thread contention will significantly slow down the algorithm. Defaults to -1.
- `random_seed` (*int*) – Seed for the random number generator. Defaults to 0.

**Attributes**

| | |
|---|---|
| **hyper-parameter_ranges** | { "penalty": ["l2"], "C": Real(0.01, 10),} |
| **model_family** | ModelFamily.LINEAR_MODEL |
| **modifies_features** | True |
| **modifies_target** | False |
| **name** | Logistic Regression Classifier |
| **supported_problem_types** | [ ProblemTypes.BINARY, ProblemTypes.MULTICLASS, ProblemTypes.TIME_SERIES_BINARY, ProblemTypes.TIME_SERIES_MULTICLASS,] |
| **training_only** | False |

**Methods**

| | |
|---|---|
| *clone* | Constructs a new component with the same parameters and random state. |
| *default_parameters* | Returns the default parameters for this component. |
| *describe* | Describe a component and its parameters. |
| *feature_importance* | Feature importance for fitted logistic regression classifier. |
| *fit* | Fits estimator to data. |
| *get_prediction_intervals* | Find the prediction intervals using the fitted regressor. |
| *load* | Loads component at file path. |
| *needs_fitting* | Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances. |
| *parameters* | Returns the parameters which were used to initialize the component. |
| *predict* | Make predictions using selected features. |
| *predict_proba* | Make probability estimates for labels. |
| *save* | Saves component at file path. |
| *update_parameters* | Updates the parameter dictionary of the component. |

**clone**(*self*)

Constructs a new component with the same parameters and random state.

**Returns** A new instance of this component with identical parameters and random state.

**default_parameters**(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

**Returns** Default parameters for this component.

**Return type** dict

**describe**(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

**Parameters**

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description as dictionary in the format {"name": name, "parameters": parameters}

**Returns** Returns dictionary if return_dict is True, else None.

**Return type** None or dict

**property feature_importance**(*self*)

Feature importance for fitted logistic regression classifier.

**fit**(*self, X: pandas.DataFrame, y: Optional[pandas.Series] = None*)

Fits estimator to data.

**Parameters**

- **X** (*pd.DataFrame*) – The input training data of shape [n_samples, n_features].
- **y** (*pd.Series, optional*) – The target training data of length [n_samples].

**Returns** self

**get_prediction_intervals**(*self, X: pandas.DataFrame, y: Optional[pandas.Series] = None, coverage: List[float] = None, predictions: pandas.Series = None*) → Dict[str, pandas.Series]

Find the prediction intervals using the fitted regressor.

This function takes the predictions of the fitted estimator and calculates the rolling standard deviation across all predictions using a window size of 5. The lower and upper predictions are determined by taking the percent point (quantile) function of the lower tail probability at each bound multiplied by the rolling standard deviation.

**Parameters**

- **X** (*pd.DataFrame*) – Data of shape [n_samples, n_features].
- **y** (*pd.Series*) – Target data. Ignored.
- **coverage** (*list[float]*) – A list of floats between the values 0 and 1 that the upper and lower bounds of the prediction interval should be calculated for.
- **predictions** (*pd.Series*) – Optional list of predictions to use. If None, will generate predictions using *X*.

**Returns** Prediction intervals, keys are in the format {coverage}_lower or {coverage}_upper.

**Return type** dict

**Raises** **MethodPropertyNotFoundError** – If the estimator does not support Time Series Regression as a problem type.

**static load**(*file_path*)

Loads component at file path.

**Parameters** **file_path** (*str*) – Location to load file.

**Returns** ComponentBase object

**needs_fitting**(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need to be fit or whose fit methods do nothing.

**Returns** True.

**property parameters**(*self*)

Returns the parameters which were used to initialize the component.

**predict**(*self*, *X: pandas.DataFrame*) → pandas.Series

Make predictions using selected features.

**Parameters** **X** (*pd.DataFrame*) – Data of shape [n_samples, n_features].

**Returns** Predicted values.

**Return type** pd.Series

**Raises** **MethodPropertyNotFoundError** – If estimator does not have a predict method or a component_obj that implements predict.

**predict_proba**(*self*, *X: pandas.DataFrame*) → pandas.Series

Make probability estimates for labels.

**Parameters** **X** (*pd.DataFrame*) – Features.

**Returns** Probability estimates.

**Return type** pd.Series

**Raises** **MethodPropertyNotFoundError** – If estimator does not have a predict_proba method or a component_obj that implements predict_proba.

**save**(*self*, *file_path*, *pickle_protocol=cloudpickle.DEFAULT_PROTOCOL*)

Saves component at file path.

**Parameters**

- **file_path** (*str*) – Location to save file.
- **pickle_protocol** (*int*) – The pickle data stream format.

**update_parameters**(*self*, *update_dict*, *reset_fit=True*)

Updates the parameter dictionary of the component.

**Parameters**

- **update_dict** (*dict*) – A dict of parameters to update.
- **reset_fit** (*bool, optional*) – If True, will set *_is_fitted* to False.

**class** evalml.pipelines.**MulticlassClassificationPipeline**(*component_graph*, *parameters=None*, *custom_name=None*, *random_seed=0*)

Pipeline subclass for all multiclass classification pipelines.

**Parameters**

- **component_graph** (ComponentGraph, *list, dict*) – ComponentGraph instance, list of components in order, or dictionary of components. Accepts strings or ComponentBase subclasses in the list. Note that when duplicate components are specified in a list, the duplicate component names will be modified with the component's index in the list. For example, the component graph [Imputer, One Hot Encoder, Imputer, Logistic Regression Classifier] will have names ["Imputer", "One Hot Encoder", "Imputer_2", "Logistic Regression Classifier"]

- **parameters** (*dict*) – Dictionary with component names as keys and dictionary of that component's parameters as values. An empty dictionary or None implies using all default values for component parameters. Defaults to None.
- **custom_name** (*str*) – Custom name for the pipeline. Defaults to None.
- **random_seed** (*int*) – Seed for the random number generator. Defaults to 0.

### Example

```
>>> pipeline = MulticlassClassificationPipeline(component_graph=["Simple Imputer",
↪"Logistic Regression Classifier"],
...                                             parameters={"Logistic Regression␣
↪Classifier": {"penalty": "elasticnet",
...                                                                                ␣
↪           "solver": "liblinear"}},
...                                             custom_name="My Multiclass Pipeline
↪")
...
>>> assert pipeline.custom_name == "My Multiclass Pipeline"
>>> assert pipeline.component_graph.component_dict.keys() == {'Simple Imputer',
↪'Logistic Regression Classifier'}
```

The pipeline parameters will be chosen from the default parameters for every component, unless specific parameters were passed in as they were above.

```
>>> assert pipeline.parameters == {
...     'Simple Imputer': {'impute_strategy': 'most_frequent', 'fill_value': None},
...     'Logistic Regression Classifier': {'penalty': 'elasticnet',
...                                        'C': 1.0,
...                                        'n_jobs': -1,
...                                        'multi_class': 'auto',
...                                        'solver': 'liblinear'}}
```

**Attributes**

| | |
|---|---|
| **problem_type** | ProblemTypes.MULTICLASS |

**Methods**

| | |
|---|---|
| *can_tune_threshold_with_objective* | Determine whether the threshold of a binary classification pipeline can be tuned. |
| *classes_* | Gets the class names for the pipeline. Will return None before pipeline is fit. |
| *clone* | Constructs a new pipeline with the same components, parameters, and random seed. |
| *create_objectives* | Create objective instances from a list of strings or objective classes. |
| *custom_name* | Custom name of the pipeline. |
| *describe* | Outputs pipeline details including component parameters. |
| *feature_importance* | Importance associated with each feature. Features dropped by the feature selection are excluded. |
| *fit* | Build a classification model. For string and categorical targets, classes are sorted by sorted(set(y)) and then are mapped to values between 0 and n_classes-1. |
| *fit_transform* | Fit and transform all components in the component graph, if all components are Transformers. |
| *get_component* | Returns component by name. |
| *get_hyperparameter_ranges* | Returns hyperparameter ranges from all components as a dictionary. |
| *graph* | Generate an image representing the pipeline graph. |
| *graph_dict* | Generates a dictionary with nodes consisting of the component names and parameters, and edges detailing component relationships. This dictionary is JSON serializable in most cases. |
| *graph_feature_importance* | Generate a bar graph of the pipeline's feature importance. |
| *inverse_transform* | Apply component inverse_transform methods to estimator predictions in reverse order. |
| *load* | Loads pipeline at file path. |
| *model_family* | Returns model family of this pipeline. |
| *name* | Name of the pipeline. |
| *new* | Constructs a new instance of the pipeline with the same component graph but with a different set of parameters. Not to be confused with python's __new__ method. |
| *parameters* | Parameter dictionary for this pipeline. |
| *predict* | Make predictions using selected features. |
| *predict_proba* | Make probability estimates for labels. |
| *save* | Saves pipeline at file path. |
| *score* | Evaluate model performance on objectives. |
| *summary* | A short summary of the pipeline structure, describing the list of components used. |
| *transform* | Transform the input. |
| *transform_all_but_final* | Transforms the data by applying all pre-processing components. |

**can_tune_threshold_with_objective**(*self*, *objective*)

Determine whether the threshold of a binary classification pipeline can be tuned.

**Parameters** **objective** (*ObjectiveBase*) – Primary AutoMLSearch objective.

**Returns** True if the pipeline threshold can be tuned.

**Return type** bool

**property classes_**(*self*)

Gets the class names for the pipeline. Will return None before pipeline is fit.

**clone**(*self*)

Constructs a new pipeline with the same components, parameters, and random seed.

**Returns** A new instance of this pipeline with identical components, parameters, and random seed.

**static create_objectives**(*objectives*)

Create objective instances from a list of strings or objective classes.

**property custom_name**(*self*)

Custom name of the pipeline.

**describe**(*self*, *return_dict=False*)

Outputs pipeline details including component parameters.

**Parameters** **return_dict** (*bool*) – If True, return dictionary of information about pipeline. Defaults to False.

**Returns** Dictionary of all component parameters if return_dict is True, else None.

**Return type** dict

**property feature_importance**(*self*)

Importance associated with each feature. Features dropped by the feature selection are excluded.

**Returns** Feature names and their corresponding importance

**Return type** pd.DataFrame

**fit**(*self*, *X*, *y*)

Build a classification model. For string and categorical targets, classes are sorted by sorted(set(y)) and then are mapped to values between 0 and n_classes-1.

**Parameters**

- **X** (*pd.DataFrame or np.ndarray*) – The input training data of shape [n_samples, n_features]
- **y** (*pd.Series, np.ndarray*) – The target training labels of length [n_samples]

**Returns** self

**Raises**

- **ValueError** – If the number of unique classes in y are not appropriate for the type of pipeline.
- **TypeError** – If the dtype is boolean but pd.NA exists in the series.
- **Exception** – For all other exceptions.

**fit_transform**(*self*, *X*, *y*)

Fit and transform all components in the component graph, if all components are Transformers.

**Parameters**

- **X** (*pd.DataFrame*) – Input features of shape [n_samples, n_features].
- **y** (*pd.Series*) – The target data of length [n_samples].

**Returns** Transformed output.

**Return type** pd.DataFrame

**Raises** **ValueError** – If final component is an Estimator.

**get_component**(*self*, *name*)

Returns component by name.

**Parameters** **name** (*str*) – Name of component.

**Returns** Component to return

**Return type** Component

**get_hyperparameter_ranges**(*self*, *custom_hyperparameters*)

Returns hyperparameter ranges from all components as a dictionary.

**Parameters** **custom_hyperparameters** (*dict*) – Custom hyperparameters for the pipeline.

**Returns** Dictionary of hyperparameter ranges for each component in the pipeline.

**Return type** dict

**graph**(*self*, *filepath=None*)

Generate an image representing the pipeline graph.

**Parameters** **filepath** (*str, optional*) – Path to where the graph should be saved. If set to None (as by default), the graph will not be saved.

**Returns** Graph object that can be directly displayed in Jupyter notebooks.

**Return type** graphviz.Digraph

**Raises**

- **RuntimeError** – If graphviz is not installed.
- **ValueError** – If path is not writeable.

**graph_dict**(*self*)

Generates a dictionary with nodes consisting of the component names and parameters, and edges detailing component relationships. This dictionary is JSON serializable in most cases.

x_edges specifies from which component feature data is being passed. y_edges specifies from which component target data is being passed. This can be used to build graphs across a variety of visualization tools. Template: {"Nodes": {"component_name": {"Name": class_name, "Parameters": parameters_attributes}, ...}}, "x_edges": [[from_component_name, to_component_name], [from_component_name, to_component_name], ...], "y_edges": [[from_component_name, to_component_name], [from_component_name, to_component_name], ...]}

**Returns** A dictionary representing the DAG structure.

**Return type** dag_dict (dict)

**graph_feature_importance**(*self*, *importance_threshold=0*)

Generate a bar graph of the pipeline's feature importance.

**Parameters** **importance_threshold** (*float, optional*) – If provided, graph features with a permutation importance whose absolute value is larger than importance_threshold. Defaults to zero.

**Returns** A bar graph showing features and their corresponding importance.

**Return type** plotly.Figure

**Raises** `ValueError` – If importance threshold is not valid.

`inverse_transform`(*self*, *y*)

Apply component inverse_transform methods to estimator predictions in reverse order.

Components that implement inverse_transform are PolynomialDecomposer, LogTransformer, LabelEncoder (tbd).

**Parameters** `y` (*pd.Series*) – Final component features.

**Returns** The inverse transform of the target.

**Return type** pd.Series

`static load`(*file_path*)

Loads pipeline at file path.

**Parameters** `file_path` (*str*) – Location to load file.

**Returns** PipelineBase object

`property model_family`(*self*)

Returns model family of this pipeline.

`property name`(*self*)

Name of the pipeline.

`new`(*self*, *parameters*, *random_seed=0*)

Constructs a new instance of the pipeline with the same component graph but with a different set of parameters. Not to be confused with python's __new__ method.

**Parameters**

- `parameters` (*dict*) – Dictionary with component names as keys and dictionary of that component's parameters as values. An empty dictionary or None implies using all default values for component parameters. Defaults to None.
- `random_seed` (*int*) – Seed for the random number generator. Defaults to 0.

**Returns** A new instance of this pipeline with identical components.

`property parameters`(*self*)

Parameter dictionary for this pipeline.

**Returns** Dictionary of all component parameters.

**Return type** dict

`predict`(*self*, *X*, *objective=None*, *X_train=None*, *y_train=None*)

Make predictions using selected features.

Note: we cast y as ints first to address boolean values that may be returned from calculating predictions which we would not be able to otherwise transform if we originally had integer targets.

**Parameters**

- `X` (*pd.DataFrame*) – Data of shape [n_samples, n_features].
- `objective` (*Object or string*) – The objective to use to make predictions.
- `X_train` (*pd.DataFrame*) – Training data. Ignored. Only used for time series.

- **y_train** (*pd.Series*) – Training labels. Ignored. Only used for time series.

**Returns** Estimated labels.

**Return type** pd.Series

**predict_proba**(*self*, *X*, *X_train=None*, *y_train=None*)

Make probability estimates for labels.

**Parameters**

- **X** (*pd.DataFrame or np.ndarray*) – Data of shape [n_samples, n_features]
- **X_train** (*pd.DataFrame or np.ndarray or None*) – Training data. Ignored. Only used for time series.
- **y_train** (*pd.Series or None*) – Training labels. Ignored. Only used for time series.

**Returns** Probability estimates

**Return type** pd.DataFrame

**Raises** **ValueError** – If final component is not an estimator.

**save**(*self*, *file_path*, *pickle_protocol=cloudpickle.DEFAULT_PROTOCOL*)

Saves pipeline at file path.

**Parameters**

- **file_path** (*str*) – Location to save file.
- **pickle_protocol** (*int*) – The pickle data stream format.

**score**(*self*, *X*, *y*, *objectives*, *X_train=None*, *y_train=None*)

Evaluate model performance on objectives.

**Parameters**

- **X** (*pd.DataFrame*) – Data of shape [n_samples, n_features]
- **y** (*pd.Series*) – True labels of length [n_samples]
- **objectives** (*list*) – List of objectives to score
- **X_train** (*pd.DataFrame*) – Training data. Ignored. Only used for time series.
- **y_train** (*pd.Series*) – Training labels. Ignored. Only used for time series.

**Returns** Ordered dictionary of objective scores.

**Return type** dict

**property summary**(*self*)

A short summary of the pipeline structure, describing the list of components used.

Example: Logistic Regression Classifier w/ Simple Imputer + One Hot Encoder

**Returns** A string describing the pipeline structure.

**transform**(*self*, *X*, *y=None*)

Transform the input.

**Parameters**

- **X** (*pd.DataFrame, or np.ndarray*) – Data of shape [n_samples, n_features].
- **y** (*pd.Series*) – The target data of length [n_samples]. Defaults to None.

**Returns** Transformed output.

**Return type** pd.DataFrame

**transform_all_but_final**(*self*, *X*, *y=None*, *X_train=None*, *y_train=None*)

Transforms the data by applying all pre-processing components.

**Parameters**

- **X** (*pd.DataFrame*) – Input data to the pipeline to transform.
- **y** (*pd.Series or None*) – Targets corresponding to X. Optional.
- **X_train** (*pd.DataFrame or np.ndarray or None*) – Training data. Only used for time series.
- **y_train** (*pd.Series or None*) – Training labels. Only used for time series.

**Returns** New transformed features.

**Return type** pd.DataFrame

**class** evalml.pipelines.**OneHotEncoder**(*top_n=10*, *features_to_encode=None*, *categories=None*, *drop='if_binary'*, *handle_unknown='ignore'*, *handle_missing='error'*, *random_seed=0*, ***kwargs*)

A transformer that encodes categorical features in a one-hot numeric array.

**Parameters**

- **top_n** (*int*) – Number of categories per column to encode. If None, all categories will be encoded. Otherwise, the *n* most frequent will be encoded and all others will be dropped. Defaults to 10.
- **features_to_encode** (*list[str]*) – List of columns to encode. All other columns will remain untouched. If None, all appropriate columns will be encoded. Defaults to None.
- **categories** (*list*) – A two dimensional list of categories, where *categories[i]* is a list of the categories for the column at index *i*. This can also be *None*, or *"auto"* if *top_n* is not None. Defaults to None.
- **drop** (*string, list*) – Method ("first" or "if_binary") to use to drop one category per feature. Can also be a list specifying which categories to drop for each feature. Defaults to 'if_binary'.
- **handle_unknown** (*string*) – Whether to ignore or error for unknown categories for a feature encountered during *fit* or *transform*. If either *top_n* or *categories* is used to limit the number of categories per column, this must be "ignore". Defaults to "ignore".
- **handle_missing** (*string*) – Options for how to handle missing (NaN) values encountered during *fit* or *transform*. If this is set to "as_category" and NaN values are within the *n* most frequent, "nan" values will be encoded as their own column. If this is set to "error", any missing values encountered will raise an error. Defaults to "error".
- **random_seed** (*int*) – Seed for the random number generator. Defaults to 0.

**Attributes**

| | |
|---|---|
| **hyper-parameter_ranges** | {} |
| **modifies_features** | True |
| **modifies_target** | False |
| **name** | One Hot Encoder |
| **training_only** | False |

**Methods**

| | |
|---|---|
| *categories* | Returns a list of the unique categories to be encoded for the particular feature, in order. |
| *clone* | Constructs a new component with the same parameters and random state. |
| *default_parameters* | Returns the default parameters for this component. |
| *describe* | Describe a component and its parameters. |
| *fit* | Fits the one-hot encoder component. |
| *fit_transform* | Fits on X and transforms X. |
| *get_feature_names* | Return feature names for the categorical features after fitting. |
| *load* | Loads component at file path. |
| *needs_fitting* | Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances. |
| *parameters* | Returns the parameters which were used to initialize the component. |
| *save* | Saves component at file path. |
| *transform* | One-hot encode the input data. |
| *update_parameters* | Updates the parameter dictionary of the component. |

**categories**(*self*, *feature_name*)

Returns a list of the unique categories to be encoded for the particular feature, in order.

**Parameters** **feature_name** (*str*) – The name of any feature provided to one-hot encoder during fit.

**Returns** The unique categories, in the same dtype as they were provided during fit.

**Return type** np.ndarray

**Raises** **ValueError** – If feature was not provided to one-hot encoder as a training feature.

**clone**(*self*)

Constructs a new component with the same parameters and random state.

**Returns** A new instance of this component with identical parameters and random state.

**default_parameters**(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

**Returns** Default parameters for this component.

**Return type** dict

**describe**(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

**Parameters**

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description as dictionary in the format {"name": name, "parameters": parameters}

**Returns** Returns dictionary if return_dict is True, else None.

**Return type** None or dict

**fit**(*self*, *X*, *y=None*)

Fits the one-hot encoder component.

**Parameters**

- **X** (*pd.DataFrame*) – The input training data of shape [n_samples, n_features].
- **y** (*pd.Series, optional*) – The target training data of length [n_samples].

**Returns** self

**Raises** **ValueError** – If encoding a column failed.

**fit_transform**(*self*, *X*, *y=None*)

Fits on X and transforms X.

**Parameters**

- **X** (*pd.DataFrame*) – Data to fit and transform.
- **y** (*pd.Series*) – Target data.

**Returns** Transformed X.

**Return type** pd.DataFrame

**Raises** **MethodPropertyNotFoundError** – If transformer does not have a transform method or a component_obj that implements transform.

**get_feature_names**(*self*)

Return feature names for the categorical features after fitting.

Feature names are formatted as {column name}_{category name}. In the event of a duplicate name, an integer will be added at the end of the feature name to distinguish it.

For example, consider a dataframe with a column called "A" and category "x_y" and another column called "A_x" with "y". In this example, the feature names would be "A_x_y" and "A_x_y_1".

**Returns** The feature names after encoding, provided in the same order as input_features.

**Return type** np.ndarray

**static load**(*file_path*)

Loads component at file path.

**Parameters** **file_path** (*str*) – Location to load file.

**Returns** ComponentBase object

**needs_fitting**(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need to be fit or whose fit methods do nothing.

**Returns** True.

**property parameters**(*self*)

Returns the parameters which were used to initialize the component.

**save**(*self*, *file_path*, *pickle_protocol=cloudpickle.DEFAULT_PROTOCOL*)

Saves component at file path.

**Parameters**

- **file_path** (*str*) – Location to save file.
- **pickle_protocol** (*int*) – The pickle data stream format.

**transform**(*self*, *X*, *y=None*)

One-hot encode the input data.

**Parameters**

- **X** (*pd.DataFrame*) – Features to one-hot encode.
- **y** (*pd.Series*) – Ignored.

**Returns** Transformed data, where each categorical feature has been encoded into numerical columns using one-hot encoding.

**Return type** pd.DataFrame

**update_parameters**(*self*, *update_dict*, *reset_fit=True*)

Updates the parameter dictionary of the component.

**Parameters**

- **update_dict** (*dict*) – A dict of parameters to update.
- **reset_fit** (*bool, optional*) – If True, will set *_is_fitted* to False.

**class** evalml.pipelines.**OrdinalEncoder**(*features_to_encode=None*, *categories=None*, *handle_unknown='error'*, *unknown_value=None*, *encoded_missing_value=None*, *random_seed=0*, ***kwargs*)

A transformer that encodes ordinal features as an array of ordinal integers representing the relative order of categories.

**Parameters**

- **features_to_encode** (*list[str]*) – List of columns to encode. All other columns will remain untouched. If None, all appropriate columns will be encoded. Defaults to None. The order of columns does not matter.
- **categories** (*dict[str, list[str]]*) – A dictionary mapping column names to their categories in the dataframes passed in at fit and transform. The order of categories specified for a column does not matter. Any category found in the data that is not present in categories will be handled as an unknown value. To not have unknown values raise an error, set handle_unknown to "use_encoded_value". Defaults to None.
- **handle_unknown** (*"error" or "use_encoded_value"*) – Whether to ignore or error for unknown categories for a feature encountered during *fit* or *transform*. When set to "error", an

error will be raised when an unknown category is found. When set to "use_encoded_value", unknown categories will be encoded as the value given for the parameter unknown_value. Defaults to "error."

- **unknown_value** (*int or np.nan*) – The value to use for unknown categories seen during fit or transform. Required when the parameter handle_unknown is set to "use_encoded_value." The value has to be distinct from the values used to encode any of the categories in fit. Defaults to None.
- **encoded_missing_value** (*int or np.nan*) – The value to use for missing (null) values seen during fit or transform. Defaults to np.nan.
- **random_seed** (*int*) – Seed for the random number generator. Defaults to 0.

**Attributes**

| | |
|---|---|
| **hyperparameter_ranges** | {} |
| **modifies_features** | True |
| **modifies_target** | False |
| **name** | Ordinal Encoder |
| **training_only** | False |

**Methods**

| | |
|---|---|
| *categories* | Returns a list of the unique categories to be encoded for the particular feature, in order. |
| *clone* | Constructs a new component with the same parameters and random state. |
| *default_parameters* | Returns the default parameters for this component. |
| *describe* | Describe a component and its parameters. |
| *fit* | Fits the ordinal encoder component. |
| *fit_transform* | Fits on X and transforms X. |
| *get_feature_names* | Return feature names for the ordinal features after fitting. |
| *load* | Loads component at file path. |
| *needs_fitting* | Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances. |
| *parameters* | Returns the parameters which were used to initialize the component. |
| *save* | Saves component at file path. |
| *transform* | Ordinally encode the input data. |
| *update_parameters* | Updates the parameter dictionary of the component. |

**categories**(*self*, *feature_name*)

Returns a list of the unique categories to be encoded for the particular feature, in order.

**Parameters** **feature_name** (*str*) – The name of any feature provided to ordinal encoder during fit.

**Returns** The unique categories, in the same dtype as they were provided during fit.

**Return type** np.ndarray

**Raises** `ValueError` – If feature was not provided to ordinal encoder as a training feature.

`clone`(*self*)

Constructs a new component with the same parameters and random state.

**Returns** A new instance of this component with identical parameters and random state.

`default_parameters`(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

**Returns** Default parameters for this component.

**Return type** dict

`describe`(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

**Parameters**

- `print_name` (*bool, optional*) – whether to print name of component
- `return_dict` (*bool, optional*) – whether to return description as dictionary in the format {"name": name, "parameters": parameters}

**Returns** Returns dictionary if return_dict is True, else None.

**Return type** None or dict

`fit`(*self*, *X*, *y=None*)

Fits the ordinal encoder component.

**Parameters**

- `X` (*pd.DataFrame*) – The input training data of shape [n_samples, n_features].
- `y` (*pd.Series, optional*) – The target training data of length [n_samples].

**Returns** self

**Raises**

- `ValueError` – If encoding a column failed.
- `TypeError` – If non-Ordinal columns are specified in features_to_encode.

`fit_transform`(*self*, *X*, *y=None*)

Fits on X and transforms X.

**Parameters**

- `X` (*pd.DataFrame*) – Data to fit and transform.
- `y` (*pd.Series*) – Target data.

**Returns** Transformed X.

**Return type** pd.DataFrame

**Raises** `MethodPropertyNotFoundError` – If transformer does not have a transform method or a component_obj that implements transform.

**get_feature_names**(*self*)

Return feature names for the ordinal features after fitting.

Feature names are formatted as {column name}_ordinal_encoding.

**Returns** The feature names after encoding, provided in the same order as input_features.

**Return type** np.ndarray

**static load**(*file_path*)

Loads component at file path.

**Parameters** **file_path** (*str*) – Location to load file.

**Returns** ComponentBase object

**needs_fitting**(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need to be fit or whose fit methods do nothing.

**Returns** True.

**property parameters**(*self*)

Returns the parameters which were used to initialize the component.

**save**(*self*, *file_path*, *pickle_protocol=cloudpickle.DEFAULT_PROTOCOL*)

Saves component at file path.

**Parameters**

- **file_path** (*str*) – Location to save file.
- **pickle_protocol** (*int*) – The pickle data stream format.

**transform**(*self*, *X*, *y=None*)

Ordinally encode the input data.

**Parameters**

- **X** (*pd.DataFrame*) – Features to encode.
- **y** (*pd.Series*) – Ignored.

**Returns** Transformed data, where each ordinal feature has been encoded into a numerical column where ordinal integers represent the relative order of categories.

**Return type** pd.DataFrame

**update_parameters**(*self*, *update_dict*, *reset_fit=True*)

Updates the parameter dictionary of the component.

**Parameters**

- **update_dict** (*dict*) – A dict of parameters to update.
- **reset_fit** (*bool, optional*) – If True, will set *_is_fitted* to False.

**class** evalml.pipelines.**PerColumnImputer**(*impute_strategies=None*, *random_seed=0*, ***kwargs*)

Imputes missing data according to a specified imputation strategy per column.

**Parameters**

- **impute_strategies** (*dict*) – Column and {"impute_strategy": strategy, "fill_value":value} pairings. Valid values for impute strategy include "mean", "median", "most_frequent", "constant" for numerical data, and "most_frequent", "constant" for object data types. Defaults to None, which uses "most_frequent" for all columns. When impute_strategy == "constant", fill_value is used to replace missing data. When None, uses 0 when imputing numerical data and "missing_value" for strings or object data types.
- **random_seed** (*int*) – Seed for the random number generator. Defaults to 0.

**Attributes**

| | |
|---|---|
| **hyper-parameter_ranges** | {} |
| **modifies_features** | True |
| **modifies_target** | False |
| **name** | Per Column Imputer |
| **training_only** | False |

**Methods**

| | |
|---|---|
| *clone* | Constructs a new component with the same parameters and random state. |
| *default_parameters* | Returns the default parameters for this component. |
| *describe* | Describe a component and its parameters. |
| *fit* | Fits imputers on input data. |
| *fit_transform* | Fits on X and transforms X. |
| *load* | Loads component at file path. |
| *needs_fitting* | Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances. |
| *parameters* | Returns the parameters which were used to initialize the component. |
| *save* | Saves component at file path. |
| *transform* | Transforms input data by imputing missing values. |
| *update_parameters* | Updates the parameter dictionary of the component. |

**clone**(*self*)

Constructs a new component with the same parameters and random state.

**Returns** A new instance of this component with identical parameters and random state.

**default_parameters**(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

**Returns** Default parameters for this component.

**Return type** dict

**describe**(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

**Parameters**

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description as dictionary in the format {"name": name, "parameters": parameters}

**Returns** Returns dictionary if return_dict is True, else None.

**Return type** None or dict

**fit**(*self*, *X*, *y=None*)

Fits imputers on input data.

**Parameters**

- **X** (*pd.DataFrame or np.ndarray*) – The input training data of shape [n_samples, n_features] to fit.
- **y** (*pd.Series, optional*) – The target training data of length [n_samples]. Ignored.

**Returns** self

**fit_transform**(*self*, *X*, *y=None*)

Fits on X and transforms X.

**Parameters**

- **X** (*pd.DataFrame*) – Data to fit and transform.
- **y** (*pd.Series*) – Target data.

**Returns** Transformed X.

**Return type** pd.DataFrame

**Raises** **MethodPropertyNotFoundError** – If transformer does not have a transform method or a component_obj that implements transform.

**static load**(*file_path*)

Loads component at file path.

**Parameters** **file_path** (*str*) – Location to load file.

**Returns** ComponentBase object

**needs_fitting**(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need to be fit or whose fit methods do nothing.

**Returns** True.

**property parameters**(*self*)

Returns the parameters which were used to initialize the component.

**save**(*self*, *file_path*, *pickle_protocol=cloudpickle.DEFAULT_PROTOCOL*)

Saves component at file path.

**Parameters**

- **file_path** (*str*) – Location to save file.
- **pickle_protocol** (*int*) – The pickle data stream format.

**transform**(*self*, *X*, *y=None*)

Transforms input data by imputing missing values.

**Parameters**

- **X** (*pd.DataFrame or np.ndarray*) – The input training data of shape [n_samples, n_features] to transform.
- **y** (*pd.Series, optional*) – The target training data of length [n_samples]. Ignored.

**Returns** Transformed X

**Return type** pd.DataFrame

**update_parameters**(*self*, *update_dict*, *reset_fit=True*)

Updates the parameter dictionary of the component.

**Parameters**

- **update_dict** (*dict*) – A dict of parameters to update.
- **reset_fit** (*bool, optional*) – If True, will set *_is_fitted* to False.

**class** evalml.pipelines.**PipelineBase**(*component_graph*, *parameters=None*, *custom_name=None*, *random_seed=0*)

Machine learning pipeline.

**Parameters**

- **component_graph** (ComponentGraph, *list, dict*) – ComponentGraph instance, list of components in order, or dictionary of components. Accepts strings or ComponentBase subclasses in the list. Note that when duplicate components are specified in a list, the duplicate component names will be modified with the component's index in the list. For example, the component graph [Imputer, One Hot Encoder, Imputer, Logistic Regression Classifier] will have names ["Imputer", "One Hot Encoder", "Imputer_2", "Logistic Regression Classifier"].
- **parameters** (*dict*) – Dictionary with component names as keys and dictionary of that component's parameters as values. An empty dictionary or None implies using all default values for component parameters. Defaults to None.
- **custom_name** (*str*) – Custom name for the pipeline. Defaults to None.
- **random_seed** (*int*) – Seed for the random number generator. Defaults to 0.

**Attributes**

| | |
|---|---|
| **problem_type** | None |

**Methods**

| | |
|---|---|
| *can_tune_threshold_with_objective* | Determine whether the threshold of a binary classification pipeline can be tuned. |
| *clone* | Constructs a new pipeline with the same components, parameters, and random seed. |
| *create_objectives* | Create objective instances from a list of strings or objective classes. |
| *custom_name* | Custom name of the pipeline. |
| *describe* | Outputs pipeline details including component parameters. |
| *feature_importance* | Importance associated with each feature. Features dropped by the feature selection are excluded. |
| *fit* | Build a model. |
| *fit_transform* | Fit and transform all components in the component graph, if all components are Transformers. |
| *get_component* | Returns component by name. |
| *get_hyperparameter_ranges* | Returns hyperparameter ranges from all components as a dictionary. |
| *graph* | Generate an image representing the pipeline graph. |
| *graph_dict* | Generates a dictionary with nodes consisting of the component names and parameters, and edges detailing component relationships. This dictionary is JSON serializable in most cases. |
| *graph_feature_importance* | Generate a bar graph of the pipeline's feature importance. |
| *inverse_transform* | Apply component inverse_transform methods to estimator predictions in reverse order. |
| *load* | Loads pipeline at file path. |
| *model_family* | Returns model family of this pipeline. |
| *name* | Name of the pipeline. |
| *new* | Constructs a new instance of the pipeline with the same component graph but with a different set of parameters. Not to be confused with python's __new__ method. |
| *parameters* | Parameter dictionary for this pipeline. |
| *predict* | Make predictions using selected features. |
| *save* | Saves pipeline at file path. |
| *score* | Evaluate model performance on current and additional objectives. |
| *summary* | A short summary of the pipeline structure, describing the list of components used. |
| *transform* | Transform the input. |
| *transform_all_but_final* | Transforms the data by applying all pre-processing components. |

**can_tune_threshold_with_objective**(*self*, *objective*)

Determine whether the threshold of a binary classification pipeline can be tuned.

**Parameters** **objective** (*ObjectiveBase*) – Primary AutoMLSearch objective.

**Returns** True if the pipeline threshold can be tuned.

**Return type** bool

**clone**(*self*)

Constructs a new pipeline with the same components, parameters, and random seed.

**Returns** A new instance of this pipeline with identical components, parameters, and random seed.

**static create_objectives**(*objectives*)

Create objective instances from a list of strings or objective classes.

**property custom_name**(*self*)

Custom name of the pipeline.

**describe**(*self*, *return_dict=False*)

Outputs pipeline details including component parameters.

**Parameters** **return_dict** (*bool*) – If True, return dictionary of information about pipeline. Defaults to False.

**Returns** Dictionary of all component parameters if return_dict is True, else None.

**Return type** dict

**property feature_importance**(*self*)

Importance associated with each feature. Features dropped by the feature selection are excluded.

**Returns** Feature names and their corresponding importance

**Return type** pd.DataFrame

**abstract fit**(*self*, *X*, *y*)

Build a model.

**Parameters**

- **X** (*pd.DataFrame or np.ndarray*) – The input training data of shape [n_samples, n_features].
- **y** (*pd.Series, np.ndarray*) – The target training data of length [n_samples].

**Returns** self

**fit_transform**(*self*, *X*, *y*)

Fit and transform all components in the component graph, if all components are Transformers.

**Parameters**

- **X** (*pd.DataFrame*) – Input features of shape [n_samples, n_features].
- **y** (*pd.Series*) – The target data of length [n_samples].

**Returns** Transformed output.

**Return type** pd.DataFrame

**Raises** **ValueError** – If final component is an Estimator.

**get_component**(*self*, *name*)

Returns component by name.

**Parameters** **name** (*str*) – Name of component.

**Returns** Component to return

**Return type** Component

**get_hyperparameter_ranges**(*self*, *custom_hyperparameters*)

Returns hyperparameter ranges from all components as a dictionary.

**Parameters** **custom_hyperparameters** (*dict*) – Custom hyperparameters for the pipeline.

**Returns** Dictionary of hyperparameter ranges for each component in the pipeline.

**Return type** dict

**graph**(*self*, *filepath=None*)

Generate an image representing the pipeline graph.

**Parameters** **filepath** (*str, optional*) – Path to where the graph should be saved. If set to None (as by default), the graph will not be saved.

**Returns** Graph object that can be directly displayed in Jupyter notebooks.

**Return type** graphviz.Digraph

**Raises**

- **RuntimeError** – If graphviz is not installed.
- **ValueError** – If path is not writeable.

**graph_dict**(*self*)

Generates a dictionary with nodes consisting of the component names and parameters, and edges detailing component relationships. This dictionary is JSON serializable in most cases.

x_edges specifies from which component feature data is being passed. y_edges specifies from which component target data is being passed. This can be used to build graphs across a variety of visualization tools. Template: {"Nodes": {"component_name": {"Name": class_name, "Parameters": parameters_attributes}, ...}}, "x_edges": [[from_component_name, to_component_name], [from_component_name, to_component_name], ...], "y_edges": [[from_component_name, to_component_name], [from_component_name, to_component_name], ...]}

**Returns** A dictionary representing the DAG structure.

**Return type** dag_dict (dict)

**graph_feature_importance**(*self*, *importance_threshold=0*)

Generate a bar graph of the pipeline's feature importance.

**Parameters** **importance_threshold** (*float, optional*) – If provided, graph features with a permutation importance whose absolute value is larger than importance_threshold. Defaults to zero.

**Returns** A bar graph showing features and their corresponding importance.

**Return type** plotly.Figure

**Raises** **ValueError** – If importance threshold is not valid.

**inverse_transform**(*self*, *y*)

Apply component inverse_transform methods to estimator predictions in reverse order.

Components that implement inverse_transform are PolynomialDecomposer, LogTransformer, LabelEncoder (tbd).

**Parameters** **y** (*pd.Series*) – Final component features.

**Returns** The inverse transform of the target.

**Return type** pd.Series

**static load**(*file_path*)

Loads pipeline at file path.

**Parameters** **file_path** (*str*) – Location to load file.

**Returns** PipelineBase object

**property model_family**(*self*)

Returns model family of this pipeline.

**property name**(*self*)

Name of the pipeline.

**new**(*self*, *parameters*, *random_seed=0*)

Constructs a new instance of the pipeline with the same component graph but with a different set of parameters. Not to be confused with python's __new__ method.

**Parameters**

- **parameters** (*dict*) – Dictionary with component names as keys and dictionary of that component's parameters as values. An empty dictionary or None implies using all default values for component parameters. Defaults to None.
- **random_seed** (*int*) – Seed for the random number generator. Defaults to 0.

**Returns** A new instance of this pipeline with identical components.

**property parameters**(*self*)

Parameter dictionary for this pipeline.

**Returns** Dictionary of all component parameters.

**Return type** dict

**predict**(*self*, *X*, *objective=None*, *X_train=None*, *y_train=None*)

Make predictions using selected features.

**Parameters**

- **X** (*pd.DataFrame, or np.ndarray*) – Data of shape [n_samples, n_features].
- **objective** (*Object or string*) – The objective to use to make predictions.
- **X_train** (*pd.DataFrame or np.ndarray or None*) – Training data. Ignored. Only used for time series.
- **y_train** (*pd.Series or None*) – Training labels. Ignored. Only used for time series.

**Returns** Predicted values.

**Return type** pd.Series

**save**(*self*, *file_path*, *pickle_protocol=cloudpickle.DEFAULT_PROTOCOL*)

Saves pipeline at file path.

**Parameters**

- **file_path** (*str*) – Location to save file.
- **pickle_protocol** (*int*) – The pickle data stream format.

**abstract score**(*self*, *X*, *y*, *objectives*, *X_train=None*, *y_train=None*)

Evaluate model performance on current and additional objectives.

**Parameters**

- **X** (*pd.DataFrame or np.ndarray*) – Data of shape [n_samples, n_features].
- **y** (*pd.Series, np.ndarray*) – True labels of length [n_samples].
- **objectives** (*list*) – Non-empty list of objectives to score on.
- **X_train** (*pd.DataFrame or np.ndarray or None*) – Training data. Ignored. Only used for time series.
- **y_train** (*pd.Series or None*) – Training labels. Ignored. Only used for time series.

**Returns** Ordered dictionary of objective scores.

**Return type** dict

**property summary**(*self*)

A short summary of the pipeline structure, describing the list of components used.

Example: Logistic Regression Classifier w/ Simple Imputer + One Hot Encoder

**Returns** A string describing the pipeline structure.

**transform**(*self*, *X*, *y=None*)

Transform the input.

**Parameters**

- **X** (*pd.DataFrame, or np.ndarray*) – Data of shape [n_samples, n_features].
- **y** (*pd.Series*) – The target data of length [n_samples]. Defaults to None.

**Returns** Transformed output.

**Return type** pd.DataFrame

**transform_all_but_final**(*self*, *X*, *y=None*, *X_train=None*, *y_train=None*)

Transforms the data by applying all pre-processing components.

**Parameters**

- **X** (*pd.DataFrame*) – Input data to the pipeline to transform.
- **y** (*pd.Series or None*) – Targets corresponding to X. Optional.
- **X_train** (*pd.DataFrame or np.ndarray or None*) – Training data. Only used for time series.
- **y_train** (*pd.Series or None*) – Training labels. Only used for time series.

**Returns** New transformed features.

**Return type** pd.DataFrame

**class** evalml.pipelines.**ProphetRegressor**(*time_index: Optional[Hashable] = None*, *changepoint_prior_scale: float = 0.05*, *seasonality_prior_scale: int = 10*, *holidays_prior_scale: int = 10*, *seasonality_mode: str = 'additive'*, *stan_backend: str = 'CMDSTANPY'*, *interval_width: float = 0.95*, *random_seed: Union[int, float] = 0*, ***kwargs*)

Prophet is a procedure for forecasting time series data based on an additive model where non-linear trends are fit with yearly, weekly, and daily seasonality, plus holiday effects. It works best with time series that have strong seasonal effects and several seasons of historical data. Prophet is robust to missing data and shifts in the trend, and typically handles outliers well.

More information here: https://facebook.github.io/prophet/

**Parameters**

- **time_index** (*str*) – Specifies the name of the column in X that provides the datetime objects. Defaults to None.
- **changepoint_prior_scale** (*float*) – Determines the strength of the sparse prior for fitting on rate changes. Increasing this value increases the flexibility of the trend. Defaults to 0.05.
- **seasonality_prior_scale** (*int*) – Similar to changepoint_prior_scale. Adjusts the extent to which the seasonality model will fit the data. Defaults to 10.
- **holidays_prior_scale** (*int*) – Similar to changepoint_prior_scale. Adjusts the extent to which holidays will fit the data. Defaults to 10.
- **seasonality_mode** (*str*) – Determines how this component fits the seasonality. Options are "additive" and "multiplicative". Defaults to "additive".
- **stan_backend** (*str*) – Determines the backend that should be used to run Prophet. Options are "CMDSTANPY" and "PYSTAN". Defaults to "CMDSTANPY".
- **interval_width** (*float*) – Determines the confidence of the prediction interval range when calling *get_prediction_intervals*. Accepts values in the range (0,1). Defaults to 0.95.
- **random_seed** (*int*) – Seed for the random number generator. Defaults to 0.

**Attributes**

| | |
|---|---|
| **hyper-parameter_ranges** | { "changepoint_prior_scale": Real(0.001, 0.5), "seasonality_prior_scale": Real(0.01, 10), "holidays_prior_scale": Real(0.01, 10), "seasonality_mode": ["additive", "multiplicative"],} |
| **model_family** | ModelFamily.PROPHET |
| **modifies_features** | True |
| **modifies_target** | False |
| **name** | Prophet Regressor |
| **supported_problem_types** | [ProblemTypes.TIME_SERIES_REGRESSION] |
| **training_only** | False |

**Methods**

| | |
|---|---|
| *build_prophet_df* | Build the Prophet data to pass fit and predict on. |
| *clone* | Constructs a new component with the same parameters and random state. |
| *default_parameters* | Returns the default parameters for this component. |
| *describe* | Describe a component and its parameters. |
| *feature_importance* | Returns array of 0's with len(1) as feature_importance is not defined for Prophet regressor. |
| *fit* | Fits Prophet regressor component to data. |
| *get_params* | Get parameters for the Prophet regressor. |
| *get_prediction_intervals* | Find the prediction intervals using the fitted ProphetRegressor. |
| *load* | Loads component at file path. |
| *needs_fitting* | Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances. |
| *parameters* | Returns the parameters which were used to initialize the component. |
| *predict* | Make predictions using fitted Prophet regressor. |
| *predict_proba* | Make probability estimates for labels. |
| *save* | Saves component at file path. |
| *update_parameters* | Updates the parameter dictionary of the component. |

**static build_prophet_df**(*X: pandas.DataFrame*, *y: Optional[pandas.Series] = None*, *time_index: str = 'ds'*) → pandas.DataFrame

Build the Prophet data to pass fit and predict on.

**clone**(*self*)

Constructs a new component with the same parameters and random state.

**Returns** A new instance of this component with identical parameters and random state.

**default_parameters**(*cls*) → dict

Returns the default parameters for this component.

**Returns** Default parameters for this component.

**Return type** dict

**describe**(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

**Parameters**

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description as dictionary in the format {"name": name, "parameters": parameters}

**Returns** Returns dictionary if return_dict is True, else None.

**Return type** None or dict

**property feature_importance**(*self*) → numpy.ndarray

Returns array of 0's with len(1) as feature_importance is not defined for Prophet regressor.

**fit**(*self*, *X: pandas.DataFrame*, *y: Optional[pandas.Series] = None*)

Fits Prophet regressor component to data.

**Parameters**

- **X** (*pd.DataFrame*) – The input training data of shape [n_samples, n_features].
- **y** (*pd.Series*) – The target training data of length [n_samples].

**Returns** self

**get_params**(*self*) → dict

Get parameters for the Prophet regressor.

**get_prediction_intervals**(*self*, *X: pandas.DataFrame*, *y: Optional[pandas.Series] = None*, *coverage: List[float] = None*, *predictions: pandas.Series = None*) → Dict[str, pandas.Series]

Find the prediction intervals using the fitted ProphetRegressor.

**Parameters**

- **X** (*pd.DataFrame*) – Data of shape [n_samples, n_features].
- **y** (*pd.Series*) – Target data. Ignored.
- **coverage** (*List[float]*) – A list of floats between the values 0 and 1 that the upper and lower bounds of the prediction interval should be calculated for.
- **predictions** (*pd.Series*) – Not used for Prophet estimator.

**Returns** Prediction intervals, keys are in the format {coverage}_lower or {coverage}_upper.

**Return type** dict

**static load**(*file_path*)

Loads component at file path.

**Parameters** **file_path** (*str*) – Location to load file.

**Returns** ComponentBase object

**needs_fitting**(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need to be fit or whose fit methods do nothing.

**Returns** True.

**property parameters**(*self*)

Returns the parameters which were used to initialize the component.

**predict**(*self*, *X: pandas.DataFrame*, *y: Optional[pandas.Series] = None*) → pandas.Series

Make predictions using fitted Prophet regressor.

**Parameters**

- **X** (*pd.DataFrame*) – Data of shape [n_samples, n_features].
- **y** (*pd.Series*) – Target data. Ignored.

**Returns** Predicted values.

**Return type** pd.Series

**predict_proba**(*self*, *X: pandas.DataFrame*) → pandas.Series

Make probability estimates for labels.

**Parameters** **X** (*pd.DataFrame*) – Features.

**Returns** Probability estimates.

**Return type** pd.Series

**Raises** **MethodPropertyNotFoundError** – If estimator does not have a predict_proba method or a component_obj that implements predict_proba.

**save**(*self*, *file_path*, *pickle_protocol=cloudpickle.DEFAULT_PROTOCOL*)

Saves component at file path.

**Parameters**

- **file_path** (*str*) – Location to save file.
- **pickle_protocol** (*int*) – The pickle data stream format.

**update_parameters**(*self*, *update_dict*, *reset_fit=True*)

Updates the parameter dictionary of the component.

**Parameters**

- **update_dict** (*dict*) – A dict of parameters to update.
- **reset_fit** (*bool, optional*) – If True, will set *_is_fitted* to False.

**class** evalml.pipelines.**RandomForestClassifier**(*n_estimators=100*, *max_depth=6*, *n_jobs=- 1*, *random_seed=0*, ***kwargs*)

Random Forest Classifier.

**Parameters**

- **n_estimators** (*float*) – The number of trees in the forest. Defaults to 100.
- **max_depth** (*int*) – Maximum tree depth for base learners. Defaults to 6.
- **n_jobs** (*int or None*) – Number of jobs to run in parallel. -1 uses all processes. Defaults to -1.
- **random_seed** (*int*) – Seed for the random number generator. Defaults to 0.

**Attributes**

| | |
|---|---|
| **hyper-parameter_ranges** | { "n_estimators": Integer(10, 1000), "max_depth": Integer(1, 10),} |
| **model_family** | ModelFamily.RANDOM_FOREST |
| **modifies_features** | True |
| **modifies_target** | False |
| **name** | Random Forest Classifier |
| **supported_problem_types** | [ ProblemTypes.BINARY, ProblemTypes.MULTICLASS, ProblemTypes.TIME_SERIES_BINARY, ProblemTypes.TIME_SERIES_MULTICLASS,] |
| **training_only** | False |

**Methods**

| | |
|---|---|
| *clone* | Constructs a new component with the same parameters and random state. |
| *default_parameters* | Returns the default parameters for this component. |
| *describe* | Describe a component and its parameters. |
| *feature_importance* | Returns importance associated with each feature. |
| *fit* | Fits estimator to data. |
| *get_prediction_intervals* | Find the prediction intervals using the fitted regressor. |
| *load* | Loads component at file path. |
| *needs_fitting* | Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances. |
| *parameters* | Returns the parameters which were used to initialize the component. |
| *predict* | Make predictions using selected features. |
| *predict_proba* | Make probability estimates for labels. |
| *save* | Saves component at file path. |
| *update_parameters* | Updates the parameter dictionary of the component. |

**clone**(*self*)

Constructs a new component with the same parameters and random state.

**Returns** A new instance of this component with identical parameters and random state.

**default_parameters**(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

**Returns** Default parameters for this component.

**Return type** dict

**describe**(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

**Parameters**

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description as dictionary in the format {"name": name, "parameters": parameters}

**Returns** Returns dictionary if return_dict is True, else None.

**Return type** None or dict

**property feature_importance**(*self*) → pandas.Series

Returns importance associated with each feature.

**Returns** Importance associated with each feature.

**Return type** np.ndarray

**Raises** **MethodPropertyNotFoundError** – If estimator does not have a feature_importance method or a component_obj that implements feature_importance.

**fit**(*self*, *X: pandas.DataFrame*, *y: Optional[pandas.Series] = None*)

Fits estimator to data.

**Parameters**

- **X** (*pd.DataFrame*) – The input training data of shape [n_samples, n_features].
- **y** (*pd.Series, optional*) – The target training data of length [n_samples].

**Returns** self

**get_prediction_intervals**(*self, X: pandas.DataFrame, y: Optional[pandas.Series] = None, coverage: List[float] = None, predictions: pandas.Series = None*) → Dict[str, pandas.Series]

Find the prediction intervals using the fitted regressor.

This function takes the predictions of the fitted estimator and calculates the rolling standard deviation across all predictions using a window size of 5. The lower and upper predictions are determined by taking the percent point (quantile) function of the lower tail probability at each bound multiplied by the rolling standard deviation.

**Parameters**

- **X** (*pd.DataFrame*) – Data of shape [n_samples, n_features].
- **y** (*pd.Series*) – Target data. Ignored.
- **coverage** (*list[float]*) – A list of floats between the values 0 and 1 that the upper and lower bounds of the prediction interval should be calculated for.
- **predictions** (*pd.Series*) – Optional list of predictions to use. If None, will generate predictions using *X*.

**Returns** Prediction intervals, keys are in the format {coverage}_lower or {coverage}_upper.

**Return type** dict

**Raises** **MethodPropertyNotFoundError** – If the estimator does not support Time Series Regression as a problem type.

**static load**(*file_path*)

Loads component at file path.

**Parameters** **file_path** (*str*) – Location to load file.

**Returns** ComponentBase object

**needs_fitting**(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need to be fit or whose fit methods do nothing.

**Returns** True.

**property parameters**(*self*)

Returns the parameters which were used to initialize the component.

**predict**(*self, X: pandas.DataFrame*) → pandas.Series

Make predictions using selected features.

**Parameters** **X** (*pd.DataFrame*) – Data of shape [n_samples, n_features].

**Returns** Predicted values.

**Return type** pd.Series

**Raises** `MethodPropertyNotFoundError` – If estimator does not have a predict method or a component_obj that implements predict.

`predict_proba`(*self*, *X: pandas.DataFrame*) → pandas.Series

Make probability estimates for labels.

**Parameters** `X` (*pd.DataFrame*) – Features.

**Returns** Probability estimates.

**Return type** pd.Series

**Raises** `MethodPropertyNotFoundError` – If estimator does not have a predict_proba method or a component_obj that implements predict_proba.

`save`(*self*, *file_path*, *pickle_protocol=cloudpickle.DEFAULT_PROTOCOL*)

Saves component at file path.

**Parameters**

- `file_path` (*str*) – Location to save file.
- `pickle_protocol` (*int*) – The pickle data stream format.

`update_parameters`(*self*, *update_dict*, *reset_fit=True*)

Updates the parameter dictionary of the component.

**Parameters**

- `update_dict` (*dict*) – A dict of parameters to update.
- `reset_fit` (*bool, optional*) – If True, will set *_is_fitted* to False.

**class** `evalml.pipelines.RandomForestRegressor`(*n_estimators: int = 100*, *max_depth: int = 6*, *n_jobs: int = - 1*, *random_seed: Union[int, float] = 0*, ***kwargs*)

Random Forest Regressor.

**Parameters**

- `n_estimators` (*float*) – The number of trees in the forest. Defaults to 100.
- `max_depth` (*int*) – Maximum tree depth for base learners. Defaults to 6.
- `n_jobs` (*int or None*) – Number of jobs to run in parallel. -1 uses all processes. Defaults to -1.
- `random_seed` (*int*) – Seed for the random number generator. Defaults to 0.

**Attributes**

| | |
|---|---|
| **hyper-parameter_ranges** | { "n_estimators": Integer(10, 1000), "max_depth": Integer(1, 32),} |
| **model_family** | ModelFamily.RANDOM_FOREST |
| **modifies_features** | True |
| **modifies_target** | False |
| **name** | Random Forest Regressor |
| **supported_problem_types** | [ ProblemTypes.REGRESSION, ProblemTypes.TIME_SERIES_REGRESSION,] |
| **training_only** | False |

**Methods**

| | |
|---|---|
| *clone* | Constructs a new component with the same parameters and random state. |
| *default_parameters* | Returns the default parameters for this component. |
| *describe* | Describe a component and its parameters. |
| *feature_importance* | Returns importance associated with each feature. |
| *fit* | Fits estimator to data. |
| *get_prediction_intervals* | Find the prediction intervals using the fitted RandomForestRegressor. |
| *load* | Loads component at file path. |
| *needs_fitting* | Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances. |
| *parameters* | Returns the parameters which were used to initialize the component. |
| *predict* | Make predictions using selected features. |
| *predict_proba* | Make probability estimates for labels. |
| *save* | Saves component at file path. |
| *update_parameters* | Updates the parameter dictionary of the component. |

**clone**(*self*)

Constructs a new component with the same parameters and random state.

**Returns** A new instance of this component with identical parameters and random state.

**default_parameters**(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

**Returns** Default parameters for this component.

**Return type** dict

**describe**(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

**Parameters**

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description as dictionary in the format {"name": name, "parameters": parameters}

**Returns** Returns dictionary if return_dict is True, else None.

**Return type** None or dict

**property feature_importance**(*self*) → pandas.Series

Returns importance associated with each feature.

**Returns** Importance associated with each feature.

**Return type** np.ndarray

**Raises** **MethodPropertyNotFoundError** – If estimator does not have a feature_importance method or a component_obj that implements feature_importance.

**fit**(*self*, *X: pandas.DataFrame*, *y: Optional[pandas.Series] = None*)

Fits estimator to data.

**Parameters**

- **X** (*pd.DataFrame*) – The input training data of shape [n_samples, n_features].
- **y** (*pd.Series, optional*) – The target training data of length [n_samples].

**Returns** self

**get_prediction_intervals**(*self*, *X: pandas.DataFrame*, *y: Optional[pandas.Series] = None*, *coverage: List[float] = None*, *predictions: pandas.Series = None*) → Dict[str, pandas.Series]

Find the prediction intervals using the fitted RandomForestRegressor.

**Parameters**

- **X** (*pd.DataFrame*) – Data of shape [n_samples, n_features].
- **y** (*pd.Series*) – Target data. Optional.
- **coverage** (*list[float]*) – A list of floats between the values 0 and 1 that the upper and lower bounds of the prediction interval should be calculated for.
- **predictions** (*pd.Series*) – Optional list of predictions to use. If None, will generate predictions using *X*.

**Returns** Prediction intervals, keys are in the format {coverage}_lower or {coverage}_upper.

**Return type** dict

**static load**(*file_path*)

Loads component at file path.

**Parameters** **file_path** (*str*) – Location to load file.

**Returns** ComponentBase object

**needs_fitting**(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need to be fit or whose fit methods do nothing.

**Returns** True.

**property parameters**(*self*)

Returns the parameters which were used to initialize the component.

**predict**(*self*, *X: pandas.DataFrame*) → pandas.Series

Make predictions using selected features.

**Parameters** **X** (*pd.DataFrame*) – Data of shape [n_samples, n_features].

**Returns** Predicted values.

**Return type** pd.Series

**Raises** **MethodPropertyNotFoundError** – If estimator does not have a predict method or a component_obj that implements predict.

**predict_proba**(*self*, *X: pandas.DataFrame*) → pandas.Series

Make probability estimates for labels.

**Parameters** **X** (*pd.DataFrame*) – Features.

**Returns** Probability estimates.

**Return type** pd.Series

**Raises** **MethodPropertyNotFoundError** – If estimator does not have a predict_proba method or a component_obj that implements predict_proba.

**save**(*self*, *file_path*, *pickle_protocol=cloudpickle.DEFAULT_PROTOCOL*)

Saves component at file path.

**Parameters**

- **file_path** (*str*) – Location to save file.
- **pickle_protocol** (*int*) – The pickle data stream format.

**update_parameters**(*self*, *update_dict*, *reset_fit=True*)

Updates the parameter dictionary of the component.

**Parameters**

- **update_dict** (*dict*) – A dict of parameters to update.
- **reset_fit** (*bool, optional*) – If True, will set *_is_fitted* to False.

**class** evalml.pipelines.**RegressionPipeline**(*component_graph*, *parameters=None*, *custom_name=None*, *random_seed=0*)

Pipeline subclass for all regression pipelines.

**Parameters**

- **component_graph** (ComponentGraph, *list, dict*) – ComponentGraph instance, list of components in order, or dictionary of components. Accepts strings or ComponentBase subclasses in the list. Note that when duplicate components are specified in a list, the duplicate component names will be modified with the component's index in the list. For example, the component graph [Imputer, One Hot Encoder, Imputer, Logistic Regression Classifier] will have names ["Imputer", "One Hot Encoder", "Imputer_2", "Logistic Regression Classifier"]
- **parameters** (*dict*) – Dictionary with component names as keys and dictionary of that component's parameters as values. An empty dictionary or None implies using all default values for component parameters. Defaults to None.
- **custom_name** (*str*) – Custom name for the pipeline. Defaults to None.
- **random_seed** (*int*) – Seed for the random number generator. Defaults to 0.

### Example

```
>>> pipeline = RegressionPipeline(component_graph=["Simple Imputer", "Linear␣
↪Regressor"],
...                               parameters={"Simple Imputer": {"impute_strategy":
↪"mean"}},
...                               custom_name="My Regression Pipeline")
...
```

(continues on next page)

(continued from previous page)

```
>>> assert pipeline.custom_name == "My Regression Pipeline"
>>> assert pipeline.component_graph.component_dict.keys() == {'Simple Imputer',
↪'Linear Regressor'}
```

The pipeline parameters will be chosen from the default parameters for every component, unless specific parameters were passed in as they were above.

```
>>> assert pipeline.parameters == {
...     'Simple Imputer': {'impute_strategy': 'mean', 'fill_value': None},
...     'Linear Regressor': {'fit_intercept': True, 'n_jobs': -1}}
```

**Attributes**

| | |
|---|---|
| **problem_type** | ProblemTypes.REGRESSION |

**Methods**

| | |
|---|---|
| *can_tune_threshold_with_objective* | Determine whether the threshold of a binary classification pipeline can be tuned. |
| *clone* | Constructs a new pipeline with the same components, parameters, and random seed. |
| *create_objectives* | Create objective instances from a list of strings or objective classes. |
| *custom_name* | Custom name of the pipeline. |
| *describe* | Outputs pipeline details including component parameters. |
| *feature_importance* | Importance associated with each feature. Features dropped by the feature selection are excluded. |
| *fit* | Build a regression model. |
| *fit_transform* | Fit and transform all components in the component graph, if all components are Transformers. |
| *get_component* | Returns component by name. |
| *get_hyperparameter_ranges* | Returns hyperparameter ranges from all components as a dictionary. |
| *graph* | Generate an image representing the pipeline graph. |
| *graph_dict* | Generates a dictionary with nodes consisting of the component names and parameters, and edges detailing component relationships. This dictionary is JSON serializable in most cases. |
| *graph_feature_importance* | Generate a bar graph of the pipeline's feature importance. |
| *inverse_transform* | Apply component inverse_transform methods to estimator predictions in reverse order. |
| *load* | Loads pipeline at file path. |
| *model_family* | Returns model family of this pipeline. |
| *name* | Name of the pipeline. |
| *new* | Constructs a new instance of the pipeline with the same component graph but with a different set of parameters. Not to be confused with python's __new__ method. |
| *parameters* | Parameter dictionary for this pipeline. |
| *predict* | Make predictions using selected features. |
| *save* | Saves pipeline at file path. |
| *score* | Evaluate model performance on current and additional objectives. |
| *summary* | A short summary of the pipeline structure, describing the list of components used. |
| *transform* | Transform the input. |
| *transform_all_but_final* | Transforms the data by applying all pre-processing components. |

**can_tune_threshold_with_objective**(*self*, *objective*)

Determine whether the threshold of a binary classification pipeline can be tuned.

**Parameters** **objective** (*ObjectiveBase*) – Primary AutoMLSearch objective.

**Returns** True if the pipeline threshold can be tuned.

**Return type** bool

**clone**(*self*)

Constructs a new pipeline with the same components, parameters, and random seed.

**Returns** A new instance of this pipeline with identical components, parameters, and random seed.

**static create_objectives**(*objectives*)

Create objective instances from a list of strings or objective classes.

**property custom_name**(*self*)

Custom name of the pipeline.

**describe**(*self*, *return_dict=False*)

Outputs pipeline details including component parameters.

**Parameters** **return_dict** (*bool*) – If True, return dictionary of information about pipeline. Defaults to False.

**Returns** Dictionary of all component parameters if return_dict is True, else None.

**Return type** dict

**property feature_importance**(*self*)

Importance associated with each feature. Features dropped by the feature selection are excluded.

**Returns** Feature names and their corresponding importance

**Return type** pd.DataFrame

**fit**(*self*, *X*, *y*)

Build a regression model.

**Parameters**

- **X** (*pd.DataFrame or np.ndarray*) – The input training data of shape [n_samples, n_features]
- **y** (*pd.Series, np.ndarray*) – The target training data of length [n_samples]

**Returns** self

**Raises** **ValueError** – If the target is not numeric.

**fit_transform**(*self*, *X*, *y*)

Fit and transform all components in the component graph, if all components are Transformers.

**Parameters**

- **X** (*pd.DataFrame*) – Input features of shape [n_samples, n_features].
- **y** (*pd.Series*) – The target data of length [n_samples].

**Returns** Transformed output.

**Return type** pd.DataFrame

**Raises** **ValueError** – If final component is an Estimator.

**get_component**(*self*, *name*)

Returns component by name.

**Parameters** **name** (*str*) – Name of component.

**Returns** Component to return

**Return type** Component

**get_hyperparameter_ranges**(*self*, *custom_hyperparameters*)

Returns hyperparameter ranges from all components as a dictionary.

**Parameters** **custom_hyperparameters** (*dict*) – Custom hyperparameters for the pipeline.

**Returns** Dictionary of hyperparameter ranges for each component in the pipeline.

**Return type** dict

**graph**(*self*, *filepath=None*)

Generate an image representing the pipeline graph.

**Parameters** **filepath** (*str, optional*) – Path to where the graph should be saved. If set to None (as by default), the graph will not be saved.

**Returns** Graph object that can be directly displayed in Jupyter notebooks.

**Return type** graphviz.Digraph

**Raises**

- **RuntimeError** – If graphviz is not installed.
- **ValueError** – If path is not writeable.

**graph_dict**(*self*)

Generates a dictionary with nodes consisting of the component names and parameters, and edges detailing component relationships. This dictionary is JSON serializable in most cases.

x_edges specifies from which component feature data is being passed. y_edges specifies from which component target data is being passed. This can be used to build graphs across a variety of visualization tools. Template: {"Nodes": {"component_name": {"Name": class_name, "Parameters": parameters_attributes}, ...}}, "x_edges": [[from_component_name, to_component_name], [from_component_name, to_component_name], ...], "y_edges": [[from_component_name, to_component_name], [from_component_name, to_component_name], ...]}

**Returns** A dictionary representing the DAG structure.

**Return type** dag_dict (dict)

**graph_feature_importance**(*self*, *importance_threshold=0*)

Generate a bar graph of the pipeline's feature importance.

**Parameters** **importance_threshold** (*float, optional*) – If provided, graph features with a permutation importance whose absolute value is larger than importance_threshold. Defaults to zero.

**Returns** A bar graph showing features and their corresponding importance.

**Return type** plotly.Figure

**Raises** **ValueError** – If importance threshold is not valid.

**inverse_transform**(*self*, *y*)

Apply component inverse_transform methods to estimator predictions in reverse order.

Components that implement inverse_transform are PolynomialDecomposer, LogTransformer, LabelEncoder (tbd).

**Parameters** **y** (*pd.Series*) – Final component features.

**Returns** The inverse transform of the target.

**Return type** pd.Series

**static load**(*file_path*)

Loads pipeline at file path.

**Parameters** **file_path** (*str*) – Location to load file.

**Returns** PipelineBase object

**property model_family**(*self*)

Returns model family of this pipeline.

**property name**(*self*)

Name of the pipeline.

**new**(*self*, *parameters*, *random_seed=0*)

Constructs a new instance of the pipeline with the same component graph but with a different set of parameters. Not to be confused with python's __new__ method.

**Parameters**

- **parameters** (*dict*) – Dictionary with component names as keys and dictionary of that component's parameters as values. An empty dictionary or None implies using all default values for component parameters. Defaults to None.
- **random_seed** (*int*) – Seed for the random number generator. Defaults to 0.

**Returns** A new instance of this pipeline with identical components.

**property parameters**(*self*)

Parameter dictionary for this pipeline.

**Returns** Dictionary of all component parameters.

**Return type** dict

**predict**(*self*, *X*, *objective=None*, *X_train=None*, *y_train=None*)

Make predictions using selected features.

**Parameters**

- **X** (*pd.DataFrame, or np.ndarray*) – Data of shape [n_samples, n_features].
- **objective** (*Object or string*) – The objective to use to make predictions.
- **X_train** (*pd.DataFrame or np.ndarray or None*) – Training data. Ignored. Only used for time series.
- **y_train** (*pd.Series or None*) – Training labels. Ignored. Only used for time series.

**Returns** Predicted values.

**Return type** pd.Series

**save**(*self*, *file_path*, *pickle_protocol=cloudpickle.DEFAULT_PROTOCOL*)

Saves pipeline at file path.

**Parameters**

- **file_path** (*str*) – Location to save file.
- **pickle_protocol** (*int*) – The pickle data stream format.

**score**(*self*, *X*, *y*, *objectives*, *X_train=None*, *y_train=None*)

Evaluate model performance on current and additional objectives.

**Parameters**

- **X** (*pd.DataFrame, or np.ndarray*) – Data of shape [n_samples, n_features]
- **y** (*pd.Series, or np.ndarray*) – True values of length [n_samples]
- **objectives** (*list*) – Non-empty list of objectives to score on
- **X_train** (*pd.DataFrame or np.ndarray or None*) – Training data. Ignored. Only used for time series.
- **y_train** (*pd.Series or None*) – Training labels. Ignored. Only used for time series.

**Returns** Ordered dictionary of objective scores.

**Return type** dict

**property summary**(*self*)

A short summary of the pipeline structure, describing the list of components used.

Example: Logistic Regression Classifier w/ Simple Imputer + One Hot Encoder

**Returns** A string describing the pipeline structure.

**transform**(*self*, *X*, *y=None*)

Transform the input.

**Parameters**

- **X** (*pd.DataFrame, or np.ndarray*) – Data of shape [n_samples, n_features].
- **y** (*pd.Series*) – The target data of length [n_samples]. Defaults to None.

**Returns** Transformed output.

**Return type** pd.DataFrame

**transform_all_but_final**(*self*, *X*, *y=None*, *X_train=None*, *y_train=None*)

Transforms the data by applying all pre-processing components.

**Parameters**

- **X** (*pd.DataFrame*) – Input data to the pipeline to transform.
- **y** (*pd.Series or None*) – Targets corresponding to X. Optional.
- **X_train** (*pd.DataFrame or np.ndarray or None*) – Training data. Only used for time series.
- **y_train** (*pd.Series or None*) – Training labels. Only used for time series.

**Returns** New transformed features.

**Return type** pd.DataFrame

**class** evalml.pipelines.**RFClassifierSelectFromModel**(*number_features=None*, *n_estimators=10*, *max_depth=None*, *percent_features=0.5*, *threshold='median'*, *n_jobs=- 1*, *random_seed=0*, ***kwargs*)

Selects top features based on importance weights using a Random Forest classifier.

**Parameters**

- **number_features** (*int*) – The maximum number of features to select. If both percent_features and number_features are specified, take the greater number of features. Defaults to None.
- **n_estimators** (*int*) – The number of trees in the forest. Defaults to 10.

- **max_depth** (*int*) – Maximum tree depth for base learners. Defaults to None.
- **percent_features** (*float*) – Percentage of features to use. If both percent_features and number_features are specified, take the greater number of features. Defaults to 0.5.
- **threshold** (*string or float*) – The threshold value to use for feature selection. Features whose importance is greater or equal are kept while the others are discarded. If "median", then the threshold value is the median of the feature importances. A scaling factor (e.g., "1.25*mean") may also be used. Defaults to median.
- **n_jobs** (*int or None*) – Number of jobs to run in parallel. -1 uses all processes. Defaults to -1.
- **random_seed** (*int*) – Seed for the random number generator. Defaults to 0.

**Attributes**

| | |
|---|---|
| **hyper-parameter_ranges** | { "percent_features": Real(0.01, 1), "threshold": ["mean", "median"],} |
| **modifies_features** | True |
| **modifies_target** | False |
| **name** | RF Classifier Select From Model |
| **training_only** | False |

**Methods**

| | |
|---|---|
| *clone* | Constructs a new component with the same parameters and random state. |
| *default_parameters* | Returns the default parameters for this component. |
| *describe* | Describe a component and its parameters. |
| *fit* | Fits component to data. |
| *fit_transform* | Fit and transform data using the feature selector. |
| *get_names* | Get names of selected features. |
| *load* | Loads component at file path. |
| *needs_fitting* | Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances. |
| *parameters* | Returns the parameters which were used to initialize the component. |
| *save* | Saves component at file path. |
| *transform* | Transforms input data by selecting features. If the component_obj does not have a transform method, will raise an MethodPropertyNotFoundError exception. |
| *update_parameters* | Updates the parameter dictionary of the component. |

**clone**(*self*)

Constructs a new component with the same parameters and random state.

**Returns** A new instance of this component with identical parameters and random state.

**default_parameters**(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

**Returns** Default parameters for this component.

**Return type** dict

**describe**(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

**Parameters**

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description as dictionary in the format {"name": name, "parameters": parameters}

**Returns** Returns dictionary if return_dict is True, else None.

**Return type** None or dict

**fit**(*self*, *X*, *y=None*)

Fits component to data.

**Parameters**

- **X** (*pd.DataFrame*) – The input training data of shape [n_samples, n_features]
- **y** (*pd.Series, optional*) – The target training data of length [n_samples]

**Returns** self

**Raises** **MethodPropertyNotFoundError** – If component does not have a fit method or a component_obj that implements fit.

**fit_transform**(*self*, *X*, *y=None*)

Fit and transform data using the feature selector.

**Parameters**

- **X** (*pd.DataFrame*) – The input training data of shape [n_samples, n_features].
- **y** (*pd.Series, optional*) – The target training data of length [n_samples].

**Returns** Transformed data.

**Return type** pd.DataFrame

**get_names**(*self*)

Get names of selected features.

**Returns** List of the names of features selected.

**Return type** list[str]

**static load**(*file_path*)

Loads component at file path.

**Parameters** **file_path** (*str*) – Location to load file.

**Returns** ComponentBase object

**needs_fitting**(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need to be fit or whose fit methods do nothing.

**Returns** True.

**property parameters**(*self*)

Returns the parameters which were used to initialize the component.

**save**(*self*, *file_path*, *pickle_protocol=cloudpickle.DEFAULT_PROTOCOL*)

Saves component at file path.

**Parameters**

- **file_path** (*str*) – Location to save file.
- **pickle_protocol** (*int*) – The pickle data stream format.

**transform**(*self*, *X*, *y=None*)

Transforms input data by selecting features. If the component_obj does not have a transform method, will raise an MethodPropertyNotFoundError exception.

**Parameters**

- **X** (*pd.DataFrame*) – Data to transform.
- **y** (*pd.Series, optional*) – Target data. Ignored.

**Returns** Transformed X

**Return type** pd.DataFrame

**Raises** **MethodPropertyNotFoundError** – If feature selector does not have a transform method or a component_obj that implements transform

**update_parameters**(*self*, *update_dict*, *reset_fit=True*)

Updates the parameter dictionary of the component.

**Parameters**

- **update_dict** (*dict*) – A dict of parameters to update.
- **reset_fit** (*bool, optional*) – If True, will set *_is_fitted* to False.

**class** evalml.pipelines.**RFRegressorSelectFromModel**(*number_features=None*, *n_estimators=10*, *max_depth=None*, *percent_features=0.5*, *threshold='median'*, *n_jobs=- 1*, *random_seed=0*, ***kwargs*)

Selects top features based on importance weights using a Random Forest regressor.

**Parameters**

- **number_features** (*int*) – The maximum number of features to select. If both percent_features and number_features are specified, take the greater number of features. Defaults to 0.5.
- **n_estimators** (*int*) – The number of trees in the forest. Defaults to 10.
- **max_depth** (*int*) – Maximum tree depth for base learners. Defaults to None.
- **percent_features** (*float*) – Percentage of features to use. If both percent_features and number_features are specified, take the greater number of features. Defaults to 0.5.

- **threshold** (*string or float*) – The threshold value to use for feature selection. Features whose importance is greater or equal are kept while the others are discarded. If "median", then the threshold value is the median of the feature importances. A scaling factor (e.g., "1.25*mean") may also be used. Defaults to median.
- **n_jobs** (*int or None*) – Number of jobs to run in parallel. -1 uses all processes. Defaults to -1.
- **random_seed** (*int*) – Seed for the random number generator. Defaults to 0.

**Attributes**

| | |
|---|---|
| **hyper-parameter_ranges** | { "percent_features": Real(0.01, 1), "threshold": ["mean", "median"],} |
| **modifies_features** | True |
| **modifies_target** | False |
| **name** | RF Regressor Select From Model |
| **training_only** | False |

**Methods**

| | |
|---|---|
| *clone* | Constructs a new component with the same parameters and random state. |
| *default_parameters* | Returns the default parameters for this component. |
| *describe* | Describe a component and its parameters. |
| *fit* | Fits component to data. |
| *fit_transform* | Fit and transform data using the feature selector. |
| *get_names* | Get names of selected features. |
| *load* | Loads component at file path. |
| *needs_fitting* | Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances. |
| *parameters* | Returns the parameters which were used to initialize the component. |
| *save* | Saves component at file path. |
| *transform* | Transforms input data by selecting features. If the component_obj does not have a transform method, will raise an MethodPropertyNotFoundError exception. |
| *update_parameters* | Updates the parameter dictionary of the component. |

**clone**(*self*)

Constructs a new component with the same parameters and random state.

**Returns** A new instance of this component with identical parameters and random state.

**default_parameters**(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

**Returns** Default parameters for this component.

**Return type** dict

`describe`(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

**Parameters**

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description as dictionary in the format {"name": name, "parameters": parameters}

**Returns** Returns dictionary if return_dict is True, else None.

**Return type** None or dict

`fit`(*self*, *X*, *y=None*)

Fits component to data.

**Parameters**

- **X** (*pd.DataFrame*) – The input training data of shape [n_samples, n_features]
- **y** (*pd.Series, optional*) – The target training data of length [n_samples]

**Returns** self

**Raises** **MethodPropertyNotFoundError** – If component does not have a fit method or a component_obj that implements fit.

`fit_transform`(*self*, *X*, *y=None*)

Fit and transform data using the feature selector.

**Parameters**

- **X** (*pd.DataFrame*) – The input training data of shape [n_samples, n_features].
- **y** (*pd.Series, optional*) – The target training data of length [n_samples].

**Returns** Transformed data.

**Return type** pd.DataFrame

`get_names`(*self*)

Get names of selected features.

**Returns** List of the names of features selected.

**Return type** list[str]

`static load`(*file_path*)

Loads component at file path.

**Parameters** **file_path** (*str*) – Location to load file.

**Returns** ComponentBase object

`needs_fitting`(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need to be fit or whose fit methods do nothing.

**Returns** True.

**property parameters**(*self*)

Returns the parameters which were used to initialize the component.

**save**(*self*, *file_path*, *pickle_protocol=cloudpickle.DEFAULT_PROTOCOL*)

Saves component at file path.

**Parameters**

- **file_path** (*str*) – Location to save file.
- **pickle_protocol** (*int*) – The pickle data stream format.

**transform**(*self*, *X*, *y=None*)

Transforms input data by selecting features. If the component_obj does not have a transform method, will raise an MethodPropertyNotFoundError exception.

**Parameters**

- **X** (*pd.DataFrame*) – Data to transform.
- **y** (*pd.Series, optional*) – Target data. Ignored.

**Returns** Transformed X

**Return type** pd.DataFrame

**Raises** **MethodPropertyNotFoundError** – If feature selector does not have a transform method or a component_obj that implements transform

**update_parameters**(*self*, *update_dict*, *reset_fit=True*)

Updates the parameter dictionary of the component.

**Parameters**

- **update_dict** (*dict*) – A dict of parameters to update.
- **reset_fit** (*bool, optional*) – If True, will set *_is_fitted* to False.

**class** evalml.pipelines.**SimpleImputer**(*impute_strategy='most_frequent'*, *fill_value=None*, *random_seed=0*, ***kwargs*)

Imputes missing data according to a specified imputation strategy. Natural language columns are ignored.

**Parameters**

- **impute_strategy** (*string*) – Impute strategy to use. Valid values include "mean", "median", "most_frequent", "constant" for numerical data, and "most_frequent", "constant" for object data types.
- **fill_value** (*string*) – When impute_strategy == "constant", fill_value is used to replace missing data. Defaults to 0 when imputing numerical data and "missing_value" for strings or object data types.
- **random_seed** (*int*) – Seed for the random number generator. Defaults to 0.

**Attributes**

| | |
|---|---|
| **hyper-parameter_ranges** | { "impute_strategy": ["mean", "median", "most_frequent"]} |
| **modifies_features** | True |
| **modifies_target** | False |
| **name** | Simple Imputer |
| **training_only** | False |

**Methods**

| | |
|---|---|
| *clone* | Constructs a new component with the same parameters and random state. |
| *default_parameters* | Returns the default parameters for this component. |
| *describe* | Describe a component and its parameters. |
| *fit* | Fits imputer to data. 'None' values are converted to np.nan before imputation and are treated as the same. |
| *fit_transform* | Fits on X and transforms X. |
| *load* | Loads component at file path. |
| *needs_fitting* | Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances. |
| *parameters* | Returns the parameters which were used to initialize the component. |
| *save* | Saves component at file path. |
| *transform* | Transforms input by imputing missing values. 'None' and np.nan values are treated as the same. |
| *update_parameters* | Updates the parameter dictionary of the component. |

**clone**(*self*)

Constructs a new component with the same parameters and random state.

**Returns** A new instance of this component with identical parameters and random state.

**default_parameters**(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

**Returns** Default parameters for this component.

**Return type** dict

**describe**(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

**Parameters**

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description as dictionary in the format {"name": name, "parameters": parameters}

**Returns** Returns dictionary if return_dict is True, else None.

**Return type** None or dict

**fit**(*self*, *X*, *y=None*)

Fits imputer to data. 'None' values are converted to np.nan before imputation and are treated as the same.

**Parameters**

- **X** (*pd.DataFrame or np.ndarray*) – the input training data of shape [n_samples, n_features]
- **y** (*pd.Series, optional*) – the target training data of length [n_samples]

**Returns** self

**Raises** **ValueError** – if the SimpleImputer receives a dataframe with both Boolean and Categorical data.

**fit_transform**(*self*, *X*, *y=None*)

Fits on X and transforms X.

**Parameters**

- **X** (*pd.DataFrame*) – Data to fit and transform
- **y** (*pd.Series, optional*) – Target data.

**Returns** Transformed X

**Return type** pd.DataFrame

**static load**(*file_path*)

Loads component at file path.

**Parameters** **file_path** (*str*) – Location to load file.

**Returns** ComponentBase object

**needs_fitting**(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need to be fit or whose fit methods do nothing.

**Returns** True.

**property parameters**(*self*)

Returns the parameters which were used to initialize the component.

**save**(*self*, *file_path*, *pickle_protocol=cloudpickle.DEFAULT_PROTOCOL*)

Saves component at file path.

**Parameters**

- **file_path** (*str*) – Location to save file.
- **pickle_protocol** (*int*) – The pickle data stream format.

**transform**(*self*, *X*, *y=None*)

Transforms input by imputing missing values. 'None' and np.nan values are treated as the same.

**Parameters**

- **X** (*pd.DataFrame*) – Data to transform.
- **y** (*pd.Series, optional*) – Ignored.

**Returns** Transformed X

**Return type** pd.DataFrame

**update_parameters**(*self*, *update_dict*, *reset_fit=True*)

Updates the parameter dictionary of the component.

**Parameters**

- **update_dict** (*dict*) – A dict of parameters to update.
- **reset_fit** (*bool, optional*) – If True, will set *_is_fitted* to False.

**class** evalml.pipelines.**StackedEnsembleBase**(*final_estimator=None*, *n_jobs=- 1*, *random_seed=0*, ***kwargs*)

Stacked Ensemble Base Class.

**Parameters**

- **final_estimator** (Estimator *or subclass*) – The estimator used to combine the base estimators.
- **n_jobs** (*int or None*) – Integer describing level of parallelism used for pipelines. None and 1 are equivalent. If set to -1, all CPUs are used. For n_jobs greater than -1, (n_cpus + 1 + n_jobs) are used. Defaults to -1. - Note: there could be some multi-process errors thrown for values of *n_jobs != 1*. If this is the case, please use *n_jobs = 1*.
- **random_seed** (*int*) – Seed for the random number generator. Defaults to 0.

**Attributes**

| | |
|---|---|
| **model_family** | ModelFamily.ENSEMBLE |
| **modifies_features** | True |
| **modifies_target** | False |
| **training_only** | False |

**Methods**

| | |
|---|---|
| *clone* | Constructs a new component with the same parameters and random state. |
| *default_parameters* | Returns the default parameters for stacked ensemble classes. |
| *describe* | Describe a component and its parameters. |
| *feature_importance* | Not implemented for StackedEnsembleClassifier and StackedEnsembleRegressor. |
| *fit* | Fits estimator to data. |
| *get_prediction_intervals* | Find the prediction intervals using the fitted regressor. |
| *load* | Loads component at file path. |
| *name* | Returns string name of this component. |
| *needs_fitting* | Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances. |
| *parameters* | Returns the parameters which were used to initialize the component. |
| *predict* | Make predictions using selected features. |
| *predict_proba* | Make probability estimates for labels. |
| *save* | Saves component at file path. |
| *supported_problem_types* | Problem types this estimator supports. |
| *update_parameters* | Updates the parameter dictionary of the component. |

**clone**(*self*)

Constructs a new component with the same parameters and random state.

**Returns** A new instance of this component with identical parameters and random state.

**default_parameters**(*cls*)

Returns the default parameters for stacked ensemble classes.

**Returns** default parameters for this component.

**Return type** dict

**describe**(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

**Parameters**

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description as dictionary in the format {"name": name, "parameters": parameters}

**Returns** Returns dictionary if return_dict is True, else None.

**Return type** None or dict

**property feature_importance**(*self*)

Not implemented for StackedEnsembleClassifier and StackedEnsembleRegressor.

**fit**(*self*, *X: pandas.DataFrame*, *y: Optional[pandas.Series] = None*)

Fits estimator to data.

**Parameters**

- **X** (*pd.DataFrame*) – The input training data of shape [n_samples, n_features].

- **y** (*pd.Series, optional*) – The target training data of length [n_samples].

**Returns** self

**get_prediction_intervals**(*self, X: pandas.DataFrame, y: Optional[pandas.Series] = None, coverage: List[float] = None, predictions: pandas.Series = None*) → Dict[str, pandas.Series]

Find the prediction intervals using the fitted regressor.

This function takes the predictions of the fitted estimator and calculates the rolling standard deviation across all predictions using a window size of 5. The lower and upper predictions are determined by taking the percent point (quantile) function of the lower tail probability at each bound multiplied by the rolling standard deviation.

**Parameters**

- **X** (*pd.DataFrame*) – Data of shape [n_samples, n_features].
- **y** (*pd.Series*) – Target data. Ignored.
- **coverage** (*list[float]*) – A list of floats between the values 0 and 1 that the upper and lower bounds of the prediction interval should be calculated for.
- **predictions** (*pd.Series*) – Optional list of predictions to use. If None, will generate predictions using *X*.

**Returns** Prediction intervals, keys are in the format {coverage}_lower or {coverage}_upper.

**Return type** dict

**Raises** **MethodPropertyNotFoundError** – If the estimator does not support Time Series Regression as a problem type.

**static load**(*file_path*)

Loads component at file path.

**Parameters** **file_path** (*str*) – Location to load file.

**Returns** ComponentBase object

**property name**(*cls*)

Returns string name of this component.

**needs_fitting**(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need to be fit or whose fit methods do nothing.

**Returns** True.

**property parameters**(*self*)

Returns the parameters which were used to initialize the component.

**predict**(*self, X: pandas.DataFrame*) → pandas.Series

Make predictions using selected features.

**Parameters** **X** (*pd.DataFrame*) – Data of shape [n_samples, n_features].

**Returns** Predicted values.

**Return type** pd.Series

**Raises** **MethodPropertyNotFoundError** – If estimator does not have a predict method or a component_obj that implements predict.

**predict_proba**(*self*, *X: pandas.DataFrame*) → pandas.Series

Make probability estimates for labels.

**Parameters** **X** (*pd.DataFrame*) – Features.

**Returns** Probability estimates.

**Return type** pd.Series

**Raises** **MethodPropertyNotFoundError** – If estimator does not have a predict_proba method or a component_obj that implements predict_proba.

**save**(*self*, *file_path*, *pickle_protocol=cloudpickle.DEFAULT_PROTOCOL*)

Saves component at file path.

**Parameters**

- **file_path** (*str*) – Location to save file.
- **pickle_protocol** (*int*) – The pickle data stream format.

**property supported_problem_types**(*cls*)

Problem types this estimator supports.

**update_parameters**(*self*, *update_dict*, *reset_fit=True*)

Updates the parameter dictionary of the component.

**Parameters**

- **update_dict** (*dict*) – A dict of parameters to update.
- **reset_fit** (*bool, optional*) – If True, will set *_is_fitted* to False.

**class** evalml.pipelines.**StackedEnsembleClassifier**(*final_estimator=None*, *n_jobs=- 1*, *random_seed=0*, ***kwargs*)

Stacked Ensemble Classifier.

**Parameters**

- **final_estimator** (*Estimator or subclass*) – The classifier used to combine the base estimators. If None, uses ElasticNetClassifier.
- **n_jobs** (*int or None*) – Integer describing level of parallelism used for pipelines. None and 1 are equivalent. If set to -1, all CPUs are used. For n_jobs below -1, (n_cpus + 1 + n_jobs) are used. Defaults to -1. - Note: there could be some multi-process errors thrown for values of *n_jobs != 1*. If this is the case, please use *n_jobs = 1*.
- **random_seed** (*int*) – Seed for the random number generator. Defaults to 0.

### Example

```
>>> from evalml.pipelines.component_graph import ComponentGraph
>>> from evalml.pipelines.components.estimators.classifiers.decision_tree_
↪classifier import DecisionTreeClassifier
>>> from evalml.pipelines.components.estimators.classifiers.elasticnet_classifier␣
↪import ElasticNetClassifier
...
>>> component_graph = {
...     "Decision Tree": [DecisionTreeClassifier(random_seed=3), "X", "y"],
...     "Decision Tree B": [DecisionTreeClassifier(random_seed=4), "X", "y"],
```

(continues on next page)

(continued from previous page)

```
...     "Stacked Ensemble": [
...         StackedEnsembleClassifier(n_jobs=1, final_
↪estimator=DecisionTreeClassifier()),
...         "Decision Tree.x",
...         "Decision Tree B.x",
...         "y",
...     ],
... }
...
>>> cg = ComponentGraph(component_graph)
>>> assert cg.default_parameters == {
...     'Decision Tree Classifier': {'criterion': 'gini',
...                                  'max_features': 'auto',
...                                  'max_depth': 6,
...                                  'min_samples_split': 2,
...                                  'min_weight_fraction_leaf': 0.0},
...     'Stacked Ensemble Classifier': {'final_estimator': ElasticNetClassifier,
...                                     'n_jobs': -1}}
```

**Attributes**

| | |
|---|---|
| **hyper-parameter_ranges** | {} |
| **model_family** | ModelFamily.ENSEMBLE |
| **modifies_features** | True |
| **modifies_target** | False |
| **name** | Stacked Ensemble Classifier |
| **supported_problem_types** | [ ProblemTypes.BINARY, ProblemTypes.MULTICLASS, ProblemTypes.TIME_SERIES_BINARY, ProblemTypes.TIME_SERIES_MULTICLASS,] |
| **training_only** | False |

**Methods**

| | |
|---|---|
| *clone* | Constructs a new component with the same parameters and random state. |
| *default_parameters* | Returns the default parameters for stacked ensemble classes. |
| *describe* | Describe a component and its parameters. |
| *feature_importance* | Not implemented for StackedEnsembleClassifier and StackedEnsembleRegressor. |
| *fit* | Fits estimator to data. |
| *get_prediction_intervals* | Find the prediction intervals using the fitted regressor. |
| *load* | Loads component at file path. |
| *needs_fitting* | Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances. |
| *parameters* | Returns the parameters which were used to initialize the component. |
| *predict* | Make predictions using selected features. |
| *predict_proba* | Make probability estimates for labels. |
| *save* | Saves component at file path. |
| *update_parameters* | Updates the parameter dictionary of the component. |

**clone**(*self*)

Constructs a new component with the same parameters and random state.

**Returns** A new instance of this component with identical parameters and random state.

**default_parameters**(*cls*)

Returns the default parameters for stacked ensemble classes.

**Returns** default parameters for this component.

**Return type** dict

**describe**(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

**Parameters**

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description as dictionary in the format {"name": name, "parameters": parameters}

**Returns** Returns dictionary if return_dict is True, else None.

**Return type** None or dict

**property feature_importance**(*self*)

Not implemented for StackedEnsembleClassifier and StackedEnsembleRegressor.

**fit**(*self*, *X: pandas.DataFrame*, *y: Optional[pandas.Series] = None*)

Fits estimator to data.

**Parameters**

- **X** (*pd.DataFrame*) – The input training data of shape [n_samples, n_features].
- **y** (*pd.Series, optional*) – The target training data of length [n_samples].

**Returns** self

**get_prediction_intervals**(*self*, *X: pandas.DataFrame*, *y: Optional[pandas.Series] = None*, *coverage: List[float] = None*, *predictions: pandas.Series = None*) → Dict[str, pandas.Series]

Find the prediction intervals using the fitted regressor.

This function takes the predictions of the fitted estimator and calculates the rolling standard deviation across all predictions using a window size of 5. The lower and upper predictions are determined by taking the percent point (quantile) function of the lower tail probability at each bound multiplied by the rolling standard deviation.

**Parameters**

- **X** (*pd.DataFrame*) – Data of shape [n_samples, n_features].
- **y** (*pd.Series*) – Target data. Ignored.
- **coverage** (*list[float]*) – A list of floats between the values 0 and 1 that the upper and lower bounds of the prediction interval should be calculated for.
- **predictions** (*pd.Series*) – Optional list of predictions to use. If None, will generate predictions using *X*.

**Returns** Prediction intervals, keys are in the format {coverage}_lower or {coverage}_upper.

**Return type** dict

**Raises** **MethodPropertyNotFoundError** – If the estimator does not support Time Series Regression as a problem type.

**static load**(*file_path*)

Loads component at file path.

**Parameters** **file_path** (*str*) – Location to load file.

**Returns** ComponentBase object

**needs_fitting**(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need to be fit or whose fit methods do nothing.

**Returns** True.

**property parameters**(*self*)

Returns the parameters which were used to initialize the component.

**predict**(*self*, *X: pandas.DataFrame*) → pandas.Series

Make predictions using selected features.

**Parameters** **X** (*pd.DataFrame*) – Data of shape [n_samples, n_features].

**Returns** Predicted values.

**Return type** pd.Series

**Raises** **MethodPropertyNotFoundError** – If estimator does not have a predict method or a component_obj that implements predict.

**predict_proba**(*self*, *X: pandas.DataFrame*) → pandas.Series

Make probability estimates for labels.

**Parameters** **X** (*pd.DataFrame*) – Features.

**Returns** Probability estimates.

**Return type** pd.Series

**Raises** **MethodPropertyNotFoundError** – If estimator does not have a predict_proba method or a component_obj that implements predict_proba.

**save**(*self*, *file_path*, *pickle_protocol=cloudpickle.DEFAULT_PROTOCOL*)

Saves component at file path.

**Parameters**

- **file_path** (*str*) – Location to save file.
- **pickle_protocol** (*int*) – The pickle data stream format.

**update_parameters**(*self*, *update_dict*, *reset_fit=True*)

Updates the parameter dictionary of the component.

**Parameters**

- **update_dict** (*dict*) – A dict of parameters to update.
- **reset_fit** (*bool, optional*) – If True, will set *_is_fitted* to False.

**class** evalml.pipelines.**StackedEnsembleRegressor**(*final_estimator=None*, *n_jobs=- 1*, *random_seed=0*, ***kwargs*)

Stacked Ensemble Regressor.

**Parameters**

- **final_estimator** (*Estimator or subclass*) – The regressor used to combine the base estimators. If None, uses ElasticNetRegressor.
- **n_jobs** (*int or None*) – Integer describing level of parallelism used for pipelines. None and 1 are equivalent. If set to -1, all CPUs are used. For n_jobs greater than -1, (n_cpus + 1 + n_jobs) are used. Defaults to -1. - Note: there could be some multi-process errors thrown for values of *n_jobs != 1*. If this is the case, please use *n_jobs = 1*.
- **random_seed** (*int*) – Seed for the random number generator. Defaults to 0.

### Example

```
>>> from evalml.pipelines.component_graph import ComponentGraph
>>> from evalml.pipelines.components.estimators.regressors.rf_regressor import␣
↪RandomForestRegressor
>>> from evalml.pipelines.components.estimators.regressors.elasticnet_regressor␣
↪import ElasticNetRegressor
...
>>> component_graph = {
...     "Random Forest": [RandomForestRegressor(random_seed=3), "X", "y"],
...     "Random Forest B": [RandomForestRegressor(random_seed=4), "X", "y"],
...     "Stacked Ensemble": [
...         StackedEnsembleRegressor(n_jobs=1, final_
↪estimator=RandomForestRegressor()),
```

(continues on next page)

(continued from previous page)

```
...         "Random Forest.x",
...         "Random Forest B.x",
...         "y",
...     ],
... }
...
>>> cg = ComponentGraph(component_graph)
>>> assert cg.default_parameters == {
...     'Random Forest Regressor': {'n_estimators': 100,
...                                 'max_depth': 6,
...                                 'n_jobs': -1},
...     'Stacked Ensemble Regressor': {'final_estimator': ElasticNetRegressor,
...                                    'n_jobs': -1}}
```

**Attributes**

| | |
|---|---|
| **hyper-parameter_ranges** | {} |
| **model_family** | ModelFamily.ENSEMBLE |
| **modifies_features** | True |
| **modifies_target** | False |
| **name** | Stacked Ensemble Regressor |
| **supported_problem_types** | [ ProblemTypes.REGRESSION, ProblemTypes.TIME_SERIES_REGRESSION,] |
| **training_only** | False |

**Methods**

| | |
|---|---|
| *clone* | Constructs a new component with the same parameters and random state. |
| *default_parameters* | Returns the default parameters for stacked ensemble classes. |
| *describe* | Describe a component and its parameters. |
| *feature_importance* | Not implemented for StackedEnsembleClassifier and StackedEnsembleRegressor. |
| *fit* | Fits estimator to data. |
| *get_prediction_intervals* | Find the prediction intervals using the fitted regressor. |
| *load* | Loads component at file path. |
| *needs_fitting* | Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances. |
| *parameters* | Returns the parameters which were used to initialize the component. |
| *predict* | Make predictions using selected features. |
| *predict_proba* | Make probability estimates for labels. |
| *save* | Saves component at file path. |
| *update_parameters* | Updates the parameter dictionary of the component. |

**clone**(*self*)

Constructs a new component with the same parameters and random state.

**Returns** A new instance of this component with identical parameters and random state.

**default_parameters**(*cls*)

Returns the default parameters for stacked ensemble classes.

**Returns** default parameters for this component.

**Return type** dict

**describe**(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

**Parameters**

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description as dictionary in the format {"name": name, "parameters": parameters}

**Returns** Returns dictionary if return_dict is True, else None.

**Return type** None or dict

**property feature_importance**(*self*)

Not implemented for StackedEnsembleClassifier and StackedEnsembleRegressor.

**fit**(*self*, *X: pandas.DataFrame*, *y: Optional[pandas.Series] = None*)

Fits estimator to data.

**Parameters**

- **X** (*pd.DataFrame*) – The input training data of shape [n_samples, n_features].
- **y** (*pd.Series, optional*) – The target training data of length [n_samples].

**Returns** self

**get_prediction_intervals**(*self*, *X: pandas.DataFrame*, *y: Optional[pandas.Series] = None*, *coverage: List[float] = None*, *predictions: pandas.Series = None*) → Dict[str, pandas.Series]

Find the prediction intervals using the fitted regressor.

This function takes the predictions of the fitted estimator and calculates the rolling standard deviation across all predictions using a window size of 5. The lower and upper predictions are determined by taking the percent point (quantile) function of the lower tail probability at each bound multiplied by the rolling standard deviation.

**Parameters**

- **X** (*pd.DataFrame*) – Data of shape [n_samples, n_features].
- **y** (*pd.Series*) – Target data. Ignored.
- **coverage** (*list[float]*) – A list of floats between the values 0 and 1 that the upper and lower bounds of the prediction interval should be calculated for.
- **predictions** (*pd.Series*) – Optional list of predictions to use. If None, will generate predictions using *X*.

**Returns** Prediction intervals, keys are in the format {coverage}_lower or {coverage}_upper.

**Return type** dict

**Raises** `MethodPropertyNotFoundError` – If the estimator does not support Time Series Regression as a problem type.

`static load`(*file_path*)

Loads component at file path.

**Parameters** `file_path` (*str*) – Location to load file.

**Returns** ComponentBase object

`needs_fitting`(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need to be fit or whose fit methods do nothing.

**Returns** True.

`property parameters`(*self*)

Returns the parameters which were used to initialize the component.

`predict`(*self*, *X: pandas.DataFrame*) → pandas.Series

Make predictions using selected features.

**Parameters** `X` (*pd.DataFrame*) – Data of shape [n_samples, n_features].

**Returns** Predicted values.

**Return type** pd.Series

**Raises** `MethodPropertyNotFoundError` – If estimator does not have a predict method or a component_obj that implements predict.

`predict_proba`(*self*, *X: pandas.DataFrame*) → pandas.Series

Make probability estimates for labels.

**Parameters** `X` (*pd.DataFrame*) – Features.

**Returns** Probability estimates.

**Return type** pd.Series

**Raises** `MethodPropertyNotFoundError` – If estimator does not have a predict_proba method or a component_obj that implements predict_proba.

`save`(*self*, *file_path*, *pickle_protocol=cloudpickle.DEFAULT_PROTOCOL*)

Saves component at file path.

**Parameters**

- `file_path` (*str*) – Location to save file.
- `pickle_protocol` (*int*) – The pickle data stream format.

`update_parameters`(*self*, *update_dict*, *reset_fit=True*)

Updates the parameter dictionary of the component.

**Parameters**

- `update_dict` (*dict*) – A dict of parameters to update.
- `reset_fit` (*bool, optional*) – If True, will set *_is_fitted* to False.

**class** evalml.pipelines.**StandardScaler**(*random_seed=0*, ***kwargs*)

A transformer that standardizes input features by removing the mean and scaling to unit variance.

**Parameters** **random_seed** (*int*) – Seed for the random number generator. Defaults to 0.

**Attributes**

| | |
|---|---|
| **hyper-parameter_ranges** | {} |
| **modifies_features** | True |
| **modifies_target** | False |
| **name** | Standard Scaler |
| **training_only** | False |

**Methods**

| | |
|---|---|
| *clone* | Constructs a new component with the same parameters and random state. |
| *default_parameters* | Returns the default parameters for this component. |
| *describe* | Describe a component and its parameters. |
| *fit* | Fits the standard scalar on the given data. |
| *fit_transform* | Fit and transform data using the standard scaler component. |
| *load* | Loads component at file path. |
| *needs_fitting* | Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances. |
| *parameters* | Returns the parameters which were used to initialize the component. |
| *save* | Saves component at file path. |
| *transform* | Transform data using the fitted standard scaler. |
| *update_parameters* | Updates the parameter dictionary of the component. |

**clone**(*self*)

Constructs a new component with the same parameters and random state.

**Returns** A new instance of this component with identical parameters and random state.

**default_parameters**(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

**Returns** Default parameters for this component.

**Return type** dict

**describe**(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

**Parameters**

- **print_name** (*bool, optional*) – whether to print name of component

- **return_dict** (*bool, optional*) – whether to return description as dictionary in the format {"name": name, "parameters": parameters}

**Returns** Returns dictionary if return_dict is True, else None.

**Return type** None or dict

**fit**(*self*, *X*, *y=None*)

Fits the standard scalar on the given data.

**Parameters**

- **X** (*pd.DataFrame*) – The input training data of shape [n_samples, n_features].
- **y** (*pd.Series, optional*) – The target training data of length [n_samples].

**Returns** self

**fit_transform**(*self*, *X*, *y=None*)

Fit and transform data using the standard scaler component.

**Parameters**

- **X** (*pd.DataFrame*) – The input training data of shape [n_samples, n_features].
- **y** (*pd.Series, optional*) – The target training data of length [n_samples].

**Returns** Transformed data.

**Return type** pd.DataFrame

**static load**(*file_path*)

Loads component at file path.

**Parameters** **file_path** (*str*) – Location to load file.

**Returns** ComponentBase object

**needs_fitting**(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need to be fit or whose fit methods do nothing.

**Returns** True.

**property parameters**(*self*)

Returns the parameters which were used to initialize the component.

**save**(*self*, *file_path*, *pickle_protocol=cloudpickle.DEFAULT_PROTOCOL*)

Saves component at file path.

**Parameters**

- **file_path** (*str*) – Location to save file.
- **pickle_protocol** (*int*) – The pickle data stream format.

**transform**(*self*, *X*, *y=None*)

Transform data using the fitted standard scaler.

**Parameters**

- **X** (*pd.DataFrame*) – The input training data of shape [n_samples, n_features].
- **y** (*pd.Series, optional*) – The target training data of length [n_samples].

**Returns** Transformed data.

**Return type** pd.DataFrame

**update_parameters**(*self*, *update_dict*, *reset_fit=True*)

Updates the parameter dictionary of the component.

**Parameters**

- **update_dict** (*dict*) – A dict of parameters to update.
- **reset_fit** (*bool, optional*) – If True, will set *_is_fitted* to False.

**class** evalml.pipelines.**SVMClassifier**(*C=1.0*, *kernel='rbf'*, *gamma='auto'*, *probability=True*, *random_seed=0*, ***kwargs*)

Support Vector Machine Classifier.

**Parameters**

- **C** (*float*) – The regularization parameter. The strength of the regularization is inversely proportional to C. Must be strictly positive. The penalty is a squared l2 penalty. Defaults to 1.0.
- **kernel** (*{"poly", "rbf", "sigmoid"}*) – Specifies the kernel type to be used in the algorithm. Defaults to "rbf".
- **gamma** (*{"scale", "auto"} or float*) – Kernel coefficient for "rbf", "poly" and "sigmoid". Defaults to "auto". - If gamma='scale' is passed then it uses 1 / (n_features * X.var()) as value of gamma - If "auto" (default), uses 1 / n_features
- **probability** (*boolean*) – Whether to enable probability estimates. Defaults to True.
- **random_seed** (*int*) – Seed for the random number generator. Defaults to 0.

**Attributes**

| | |
|---|---|
| **hyper-parameter_ranges** | { "C": Real(0, 10), "kernel": ["poly", "rbf", "sigmoid"], "gamma": ["scale", "auto"],} |
| **model_family** | ModelFamily.SVM |
| **modifies_features** | True |
| **modifies_target** | False |
| **name** | SVM Classifier |
| **supported_problem_types** | [ ProblemTypes.BINARY, ProblemTypes.MULTICLASS, ProblemTypes.TIME_SERIES_BINARY, ProblemTypes.TIME_SERIES_MULTICLASS,] |
| **training_only** | False |

**Methods**

| | |
|---|---|
| *clone* | Constructs a new component with the same parameters and random state. |
| *default_parameters* | Returns the default parameters for this component. |
| *describe* | Describe a component and its parameters. |
| *feature_importance* | Feature importance only works with linear kernels. |
| *fit* | Fits estimator to data. |
| *get_prediction_intervals* | Find the prediction intervals using the fitted regressor. |
| *load* | Loads component at file path. |
| *needs_fitting* | Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances. |
| *parameters* | Returns the parameters which were used to initialize the component. |
| *predict* | Make predictions using selected features. |
| *predict_proba* | Make probability estimates for labels. |
| *save* | Saves component at file path. |
| *update_parameters* | Updates the parameter dictionary of the component. |

**clone**(*self*)

Constructs a new component with the same parameters and random state.

**Returns** A new instance of this component with identical parameters and random state.

**default_parameters**(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

**Returns** Default parameters for this component.

**Return type** dict

**describe**(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

**Parameters**

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description as dictionary in the format {"name": name, "parameters": parameters}

**Returns** Returns dictionary if return_dict is True, else None.

**Return type** None or dict

**property feature_importance**(*self*)

Feature importance only works with linear kernels.

If the kernel isn't linear, we return a numpy array of zeros.

**Returns** Feature importance of fitted SVM classifier or a numpy array of zeroes if the kernel is not linear.

**fit**(*self*, *X: pandas.DataFrame*, *y: Optional[pandas.Series] = None*)

Fits estimator to data.

**Parameters**

- **X** (*pd.DataFrame*) – The input training data of shape [n_samples, n_features].
- **y** (*pd.Series, optional*) – The target training data of length [n_samples].

**Returns** self

**get_prediction_intervals**(*self, X: pandas.DataFrame, y: Optional[pandas.Series] = None, coverage: List[float] = None, predictions: pandas.Series = None*) → Dict[str, pandas.Series]

Find the prediction intervals using the fitted regressor.

This function takes the predictions of the fitted estimator and calculates the rolling standard deviation across all predictions using a window size of 5. The lower and upper predictions are determined by taking the percent point (quantile) function of the lower tail probability at each bound multiplied by the rolling standard deviation.

**Parameters**

- **X** (*pd.DataFrame*) – Data of shape [n_samples, n_features].
- **y** (*pd.Series*) – Target data. Ignored.
- **coverage** (*list[float]*) – A list of floats between the values 0 and 1 that the upper and lower bounds of the prediction interval should be calculated for.
- **predictions** (*pd.Series*) – Optional list of predictions to use. If None, will generate predictions using *X*.

**Returns** Prediction intervals, keys are in the format {coverage}_lower or {coverage}_upper.

**Return type** dict

**Raises** **MethodPropertyNotFoundError** – If the estimator does not support Time Series Regression as a problem type.

**static load**(*file_path*)

Loads component at file path.

**Parameters** **file_path** (*str*) – Location to load file.

**Returns** ComponentBase object

**needs_fitting**(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need to be fit or whose fit methods do nothing.

**Returns** True.

**property parameters**(*self*)

Returns the parameters which were used to initialize the component.

**predict**(*self, X: pandas.DataFrame*) → pandas.Series

Make predictions using selected features.

**Parameters** **X** (*pd.DataFrame*) – Data of shape [n_samples, n_features].

**Returns** Predicted values.

**Return type** pd.Series

**Raises** **MethodPropertyNotFoundError** – If estimator does not have a predict method or a component_obj that implements predict.

**predict_proba**(*self*, *X: pandas.DataFrame*) → pandas.Series

Make probability estimates for labels.

**Parameters** **X** (*pd.DataFrame*) – Features.

**Returns** Probability estimates.

**Return type** pd.Series

**Raises** **MethodPropertyNotFoundError** – If estimator does not have a predict_proba method or a component_obj that implements predict_proba.

**save**(*self*, *file_path*, *pickle_protocol=cloudpickle.DEFAULT_PROTOCOL*)

Saves component at file path.

**Parameters**

- **file_path** (*str*) – Location to save file.
- **pickle_protocol** (*int*) – The pickle data stream format.

**update_parameters**(*self*, *update_dict*, *reset_fit=True*)

Updates the parameter dictionary of the component.

**Parameters**

- **update_dict** (*dict*) – A dict of parameters to update.
- **reset_fit** (*bool, optional*) – If True, will set *_is_fitted* to False.

**class** evalml.pipelines.**SVMRegressor**(*C=1.0*, *kernel='rbf'*, *gamma='auto'*, *random_seed=0*, ***kwargs*)

Support Vector Machine Regressor.

**Parameters**

- **C** (*float*) – The regularization parameter. The strength of the regularization is inversely proportional to C. Must be strictly positive. The penalty is a squared l2 penalty. Defaults to 1.0.
- **kernel** (*{"poly", "rbf", "sigmoid"}*) – Specifies the kernel type to be used in the algorithm. Defaults to "rbf".
- **gamma** (*{"scale", "auto"} or float*) – Kernel coefficient for "rbf", "poly" and "sigmoid". Defaults to "auto". - If gamma='scale' is passed then it uses 1 / (n_features * X.var()) as value of gamma - If "auto" (default), uses 1 / n_features
- **random_seed** (*int*) – Seed for the random number generator. Defaults to 0.

**Attributes**

| | |
|---|---|
| **hyper-parameter_ranges** | { "C": Real(0, 10), "kernel": ["poly", "rbf", "sigmoid"], "gamma": ["scale", "auto"],} |
| **model_family** | ModelFamily.SVM |
| **modifies_features** | True |
| **modifies_target** | False |
| **name** | SVM Regressor |
| **supported_problem_types** | [ ProblemTypes.REGRESSION, ProblemTypes.TIME_SERIES_REGRESSION,] |
| **training_only** | False |

**Methods**

| | |
|---|---|
| *clone* | Constructs a new component with the same parameters and random state. |
| *default_parameters* | Returns the default parameters for this component. |
| *describe* | Describe a component and its parameters. |
| *feature_importance* | Feature importance of fitted SVM regresor. |
| *fit* | Fits estimator to data. |
| *get_prediction_intervals* | Find the prediction intervals using the fitted regressor. |
| *load* | Loads component at file path. |
| *needs_fitting* | Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances. |
| *parameters* | Returns the parameters which were used to initialize the component. |
| *predict* | Make predictions using selected features. |
| *predict_proba* | Make probability estimates for labels. |
| *save* | Saves component at file path. |
| *update_parameters* | Updates the parameter dictionary of the component. |

**clone**(*self*)

Constructs a new component with the same parameters and random state.

**Returns** A new instance of this component with identical parameters and random state.

**default_parameters**(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

**Returns** Default parameters for this component.

**Return type** dict

**describe**(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

**Parameters**

- **print_name** (*bool, optional*) – whether to print name of component

- **return_dict** (*bool, optional*) – whether to return description as dictionary in the format {"name": name, "parameters": parameters}

**Returns** Returns dictionary if return_dict is True, else None.

**Return type** None or dict

**property feature_importance**(*self*)

Feature importance of fitted SVM regresor.

Only works with linear kernels. If the kernel isn't linear, we return a numpy array of zeros.

**Returns** The feature importance of the fitted SVM regressor, or an array of zeroes if the kernel is not linear.

**fit**(*self*, *X: pandas.DataFrame*, *y: Optional[pandas.Series] = None*)

Fits estimator to data.

**Parameters**

- **X** (*pd.DataFrame*) – The input training data of shape [n_samples, n_features].
- **y** (*pd.Series, optional*) – The target training data of length [n_samples].

**Returns** self

**get_prediction_intervals**(*self*, *X: pandas.DataFrame*, *y: Optional[pandas.Series] = None*, *coverage: List[float] = None*, *predictions: pandas.Series = None*) → Dict[str, pandas.Series]

Find the prediction intervals using the fitted regressor.

This function takes the predictions of the fitted estimator and calculates the rolling standard deviation across all predictions using a window size of 5. The lower and upper predictions are determined by taking the percent point (quantile) function of the lower tail probability at each bound multiplied by the rolling standard deviation.

**Parameters**

- **X** (*pd.DataFrame*) – Data of shape [n_samples, n_features].
- **y** (*pd.Series*) – Target data. Ignored.
- **coverage** (*list[float]*) – A list of floats between the values 0 and 1 that the upper and lower bounds of the prediction interval should be calculated for.
- **predictions** (*pd.Series*) – Optional list of predictions to use. If None, will generate predictions using *X*.

**Returns** Prediction intervals, keys are in the format {coverage}_lower or {coverage}_upper.

**Return type** dict

**Raises** **MethodPropertyNotFoundError** – If the estimator does not support Time Series Regression as a problem type.

**static load**(*file_path*)

Loads component at file path.

**Parameters** **file_path** (*str*) – Location to load file.

**Returns** ComponentBase object

**needs_fitting**(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need to be fit or whose fit methods do nothing.

**Returns** True.

**property parameters**(*self*)

Returns the parameters which were used to initialize the component.

**predict**(*self*, *X: pandas.DataFrame*) → pandas.Series

Make predictions using selected features.

**Parameters** **X** (*pd.DataFrame*) – Data of shape [n_samples, n_features].

**Returns** Predicted values.

**Return type** pd.Series

**Raises** **MethodPropertyNotFoundError** – If estimator does not have a predict method or a component_obj that implements predict.

**predict_proba**(*self*, *X: pandas.DataFrame*) → pandas.Series

Make probability estimates for labels.

**Parameters** **X** (*pd.DataFrame*) – Features.

**Returns** Probability estimates.

**Return type** pd.Series

**Raises** **MethodPropertyNotFoundError** – If estimator does not have a predict_proba method or a component_obj that implements predict_proba.

**save**(*self*, *file_path*, *pickle_protocol=cloudpickle.DEFAULT_PROTOCOL*)

Saves component at file path.

**Parameters**

- **file_path** (*str*) – Location to save file.
- **pickle_protocol** (*int*) – The pickle data stream format.

**update_parameters**(*self*, *update_dict*, *reset_fit=True*)

Updates the parameter dictionary of the component.

**Parameters**

- **update_dict** (*dict*) – A dict of parameters to update.
- **reset_fit** (*bool, optional*) – If True, will set *_is_fitted* to False.

**class** evalml.pipelines.**TargetEncoder**(*cols=None*, *smoothing=1*, *handle_unknown='value'*, *handle_missing='value'*, *random_seed=0*, ***kwargs*)

A transformer that encodes categorical features into target encodings.

**Parameters**

- **cols** (*list*) – Columns to encode. If None, all string columns will be encoded, otherwise only the columns provided will be encoded. Defaults to None
- **smoothing** (*float*) – The smoothing factor to apply. The larger this value is, the more influence the expected target value has on the resulting target encodings. Must be strictly larger than 0. Defaults to 1.0

- **handle_unknown** (*string*) – Determines how to handle unknown categories for a feature encountered. Options are 'value', 'error', nd 'return_nan'. Defaults to 'value', which replaces with the target mean
- **handle_missing** (*string*) – Determines how to handle missing values encountered during *fit* or *transform*. Options are 'value', 'error', and 'return_nan'. Defaults to 'value', which replaces with the target mean
- **random_seed** (*int*) – Seed for the random number generator. Defaults to 0.

**Attributes**

| | |
|---|---|
| **hyper-parameter_ranges** | {} |
| **modifies_features** | True |
| **modifies_target** | False |
| **name** | Target Encoder |
| **training_only** | False |

**Methods**

| | |
|---|---|
| *clone* | Constructs a new component with the same parameters and random state. |
| *default_parameters* | Returns the default parameters for this component. |
| *describe* | Describe a component and its parameters. |
| *fit* | Fits the target encoder. |
| *fit_transform* | Fit and transform data using the target encoder. |
| *get_feature_names* | Return feature names for the input features after fitting. |
| *load* | Loads component at file path. |
| *needs_fitting* | Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances. |
| *parameters* | Returns the parameters which were used to initialize the component. |
| *save* | Saves component at file path. |
| *transform* | Transform data using the fitted target encoder. |
| *update_parameters* | Updates the parameter dictionary of the component. |

**clone**(*self*)

Constructs a new component with the same parameters and random state.

**Returns** A new instance of this component with identical parameters and random state.

**default_parameters**(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

**Returns** Default parameters for this component.

**Return type** dict

**describe**(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

**Parameters**

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description as dictionary in the format {"name": name, "parameters": parameters}

**Returns** Returns dictionary if return_dict is True, else None.

**Return type** None or dict

**fit**(*self*, *X*, *y*)

Fits the target encoder.

**Parameters**

- **X** (*pd.DataFrame*) – The input training data of shape [n_samples, n_features].
- **y** (*pd.Series, optional*) – The target training data of length [n_samples].

**Returns** self

**fit_transform**(*self*, *X*, *y*)

Fit and transform data using the target encoder.

**Parameters**

- **X** (*pd.DataFrame*) – The input training data of shape [n_samples, n_features].
- **y** (*pd.Series, optional*) – The target training data of length [n_samples].

**Returns** Transformed data.

**Return type** pd.DataFrame

**get_feature_names**(*self*)

Return feature names for the input features after fitting.

**Returns** The feature names after encoding.

**Return type** np.array

**static load**(*file_path*)

Loads component at file path.

**Parameters** **file_path** (*str*) – Location to load file.

**Returns** ComponentBase object

**needs_fitting**(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need to be fit or whose fit methods do nothing.

**Returns** True.

**property parameters**(*self*)

Returns the parameters which were used to initialize the component.

**save**(*self*, *file_path*, *pickle_protocol=cloudpickle.DEFAULT_PROTOCOL*)

Saves component at file path.

**Parameters**

- **file_path** (*str*) – Location to save file.
- **pickle_protocol** (*int*) – The pickle data stream format.

**transform**(*self*, *X*, *y=None*)

Transform data using the fitted target encoder.

**Parameters**

- **X** (*pd.DataFrame*) – The input training data of shape [n_samples, n_features].
- **y** (*pd.Series, optional*) – The target training data of length [n_samples].

**Returns** Transformed data.

**Return type** pd.DataFrame

**update_parameters**(*self*, *update_dict*, *reset_fit=True*)

Updates the parameter dictionary of the component.

**Parameters**

- **update_dict** (*dict*) – A dict of parameters to update.
- **reset_fit** (*bool, optional*) – If True, will set *_is_fitted* to False.

**class** evalml.pipelines.**TimeSeriesBinaryClassificationPipeline**(*component_graph*, *parameters=None*, *custom_name=None*, *random_seed=0*)

Pipeline base class for time series binary classification problems.

**Parameters**

- **component_graph** (*list or dict*) – List of components in order. Accepts strings or ComponentBase subclasses in the list. Note that when duplicate components are specified in a list, the duplicate component names will be modified with the component's index in the list. For example, the component graph [Imputer, One Hot Encoder, Imputer, Logistic Regression Classifier] will have names ["Imputer", "One Hot Encoder", "Imputer_2", "Logistic Regression Classifier"]
- **parameters** (*dict*) – Dictionary with component names as keys and dictionary of that component's parameters as values. An empty dictionary {} implies using all default values for component parameters. Pipeline-level parameters such as time_index, gap, and max_delay must be specified with the "pipeline" key. For example: Pipeline(parameters={"pipeline": {"time_index": "Date", "max_delay": 4, "gap": 2}}).
- **random_seed** (*int*) – Seed for the random number generator. Defaults to 0.

### Example

```
>>> pipeline = TimeSeriesBinaryClassificationPipeline(component_graph=["Simple␣
↪Imputer", "Logistic Regression Classifier"],
...                                                    parameters={"Logistic␣
↪Regression Classifier": {"penalty": "elasticnet",
...                                                                                ␣
↪                 "solver": "liblinear"},
...                                                                 "pipeline": {"gap
↪": 1, "max_delay": 1, "forecast_horizon": 1, "time_index": "date"}},
...                                                    custom_name="My␣
↪TimeSeriesBinary Pipeline")
...
>>> assert pipeline.custom_name == "My TimeSeriesBinary Pipeline"
>>> assert pipeline.component_graph.component_dict.keys() == {'Simple Imputer',
↪'Logistic Regression Classifier'}
...
>>> assert pipeline.parameters == {
...     'Simple Imputer': {'impute_strategy': 'most_frequent', 'fill_value': None},
...     'Logistic Regression Classifier': {'penalty': 'elasticnet',
...                                        'C': 1.0,
...                                        'n_jobs': -1,
...                                        'multi_class': 'auto',
...                                        'solver': 'liblinear'},
...     'pipeline': {'gap': 1, 'max_delay': 1, 'forecast_horizon': 1, 'time_index':
↪"date"}}
```

**Attributes**

| | |
|---|---|
| **problem_type** | None |

**Methods**

| | |
|---|---|
| *can_tune_threshold_with_objective* | Determine whether the threshold of a binary classification pipeline can be tuned. |
| *classes_* | Gets the class names for the pipeline. Will return None before pipeline is fit. |
| *clone* | Constructs a new pipeline with the same components, parameters, and random seed. |
| *create_objectives* | Create objective instances from a list of strings or objective classes. |
| *custom_name* | Custom name of the pipeline. |
| *dates_needed_for_prediction* | Return dates needed to forecast the given date in the future. |
| *dates_needed_for_prediction_range* | Return dates needed to forecast the given date in the future. |
| *describe* | Outputs pipeline details including component parameters. |
| *feature_importance* | Importance associated with each feature. Features dropped by the feature selection are excluded. |
| *fit* | Fit a time series classification model. |

continues on next page

Table 12 – continued from previous page

| | |
|---|---|
| *fit_transform* | Fit and transform all components in the component graph, if all components are Transformers. |
| *get_component* | Returns component by name. |
| *get_hyperparameter_ranges* | Returns hyperparameter ranges from all components as a dictionary. |
| *graph* | Generate an image representing the pipeline graph. |
| *graph_dict* | Generates a dictionary with nodes consisting of the component names and parameters, and edges detailing component relationships. This dictionary is JSON serializable in most cases. |
| *graph_feature_importance* | Generate a bar graph of the pipeline's feature importance. |
| *inverse_transform* | Apply component inverse_transform methods to estimator predictions in reverse order. |
| *load* | Loads pipeline at file path. |
| *model_family* | Returns model family of this pipeline. |
| *name* | Name of the pipeline. |
| *new* | Constructs a new instance of the pipeline with the same component graph but with a different set of parameters. Not to be confused with python's __new__ method. |
| *optimize_threshold* | Optimize the pipeline threshold given the objective to use. Only used for binary problems with objectives whose thresholds can be tuned. |
| *parameters* | Parameter dictionary for this pipeline. |
| *predict* | Predict on future data where target is not known. |
| *predict_in_sample* | Predict on future data where the target is known, e.g. cross validation. |
| *predict_proba* | Predict on future data where the target is unknown. |
| *predict_proba_in_sample* | Predict on future data where the target is known, e.g. cross validation. |
| *save* | Saves pipeline at file path. |
| *score* | Evaluate model performance on current and additional objectives. |
| *summary* | A short summary of the pipeline structure, describing the list of components used. |
| *threshold* | Threshold used to make a prediction. Defaults to None. |
| *transform* | Transform the input. |
| *transform_all_but_final* | Transforms the data by applying all pre-processing components. |

**can_tune_threshold_with_objective**(*self*, *objective*)

Determine whether the threshold of a binary classification pipeline can be tuned.

**Parameters** **objective** (*ObjectiveBase*) – Primary AutoMLSearch objective.

**Returns** True if the pipeline threshold can be tuned.

**Return type** bool

**property classes_**(*self*)

Gets the class names for the pipeline. Will return None before pipeline is fit.

**clone**(*self*)

Constructs a new pipeline with the same components, parameters, and random seed.

**Returns** A new instance of this pipeline with identical components, parameters, and random seed.

**static create_objectives**(*objectives*)

Create objective instances from a list of strings or objective classes.

**property custom_name**(*self*)

Custom name of the pipeline.

**dates_needed_for_prediction**(*self*, *date*)

Return dates needed to forecast the given date in the future.

**Parameters** **date** (*pd.Timestamp*) – Date to forecast in the future.

**Returns** Range of dates needed to forecast the given date.

**Return type** dates_needed (tuple(pd.Timestamp))

**dates_needed_for_prediction_range**(*self*, *start_date*, *end_date*)

Return dates needed to forecast the given date in the future.

**Parameters**

- **start_date** (*pd.Timestamp*) – Start date of range to forecast in the future.
- **end_date** (*pd.Timestamp*) – End date of range to forecast in the future.

**Returns** Range of dates needed to forecast the given date.

**Return type** dates_needed (tuple(pd.Timestamp))

**Raises** **ValueError** – If start_date doesn't come before end_date

**describe**(*self*, *return_dict=False*)

Outputs pipeline details including component parameters.

**Parameters** **return_dict** (*bool*) – If True, return dictionary of information about pipeline. Defaults to False.

**Returns** Dictionary of all component parameters if return_dict is True, else None.

**Return type** dict

**property feature_importance**(*self*)

Importance associated with each feature. Features dropped by the feature selection are excluded.

**Returns** Feature names and their corresponding importance

**Return type** pd.DataFrame

**fit**(*self*, *X*, *y*)

Fit a time series classification model.

**Parameters**

- **X** (*pd.DataFrame or np.ndarray*) – The input training data of shape [n_samples, n_features]
- **y** (*pd.Series, np.ndarray*) – The target training labels of length [n_samples]

**Returns** self

**Raises** `ValueError` – If the number of unique classes in y are not appropriate for the type of pipeline.

`fit_transform`(*self*, *X*, *y*)

Fit and transform all components in the component graph, if all components are Transformers.

**Parameters**

- `X` (*pd.DataFrame*) – Input features of shape [n_samples, n_features].
- `y` (*pd.Series*) – The target data of length [n_samples].

**Returns** Transformed output.

**Return type** pd.DataFrame

**Raises** `ValueError` – If final component is an Estimator.

`get_component`(*self*, *name*)

Returns component by name.

**Parameters** `name` (*str*) – Name of component.

**Returns** Component to return

**Return type** Component

`get_hyperparameter_ranges`(*self*, *custom_hyperparameters*)

Returns hyperparameter ranges from all components as a dictionary.

**Parameters** `custom_hyperparameters` (*dict*) – Custom hyperparameters for the pipeline.

**Returns** Dictionary of hyperparameter ranges for each component in the pipeline.

**Return type** dict

`graph`(*self*, *filepath=None*)

Generate an image representing the pipeline graph.

**Parameters** `filepath` (*str, optional*) – Path to where the graph should be saved. If set to None (as by default), the graph will not be saved.

**Returns** Graph object that can be directly displayed in Jupyter notebooks.

**Return type** graphviz.Digraph

**Raises**

- `RuntimeError` – If graphviz is not installed.
- `ValueError` – If path is not writeable.

`graph_dict`(*self*)

Generates a dictionary with nodes consisting of the component names and parameters, and edges detailing component relationships. This dictionary is JSON serializable in most cases.

x_edges specifies from which component feature data is being passed. y_edges specifies from which component target data is being passed. This can be used to build graphs across a variety of visualization tools. Template: {"Nodes": {"component_name": {"Name": class_name, "Parameters": parameters_attributes}, ...}}, "x_edges": [[from_component_name, to_component_name], [from_component_name, to_component_name], ...], "y_edges": [[from_component_name, to_component_name], [from_component_name, to_component_name], ...]}

**Returns** A dictionary representing the DAG structure.

**Return type** dag_dict (dict)

**graph_feature_importance**(*self*, *importance_threshold=0*)

Generate a bar graph of the pipeline's feature importance.

**Parameters** **importance_threshold** (*float, optional*) – If provided, graph features with a permutation importance whose absolute value is larger than importance_threshold. Defaults to zero.

**Returns** A bar graph showing features and their corresponding importance.

**Return type** plotly.Figure

**Raises** **ValueError** – If importance threshold is not valid.

**inverse_transform**(*self*, *y*)

Apply component inverse_transform methods to estimator predictions in reverse order.

Components that implement inverse_transform are PolynomialDecomposer, LogTransformer, LabelEncoder (tbd).

**Parameters** **y** (*pd.Series*) – Final component features.

**Returns** The inverse transform of the target.

**Return type** pd.Series

**static load**(*file_path*)

Loads pipeline at file path.

**Parameters** **file_path** (*str*) – Location to load file.

**Returns** PipelineBase object

**property model_family**(*self*)

Returns model family of this pipeline.

**property name**(*self*)

Name of the pipeline.

**new**(*self*, *parameters*, *random_seed=0*)

Constructs a new instance of the pipeline with the same component graph but with a different set of parameters. Not to be confused with python's __new__ method.

**Parameters**

- **parameters** (*dict*) – Dictionary with component names as keys and dictionary of that component's parameters as values. An empty dictionary or None implies using all default values for component parameters. Defaults to None.
- **random_seed** (*int*) – Seed for the random number generator. Defaults to 0.

**Returns** A new instance of this pipeline with identical components.

**optimize_threshold**(*self*, *X*, *y*, *y_pred_proba*, *objective*)

Optimize the pipeline threshold given the objective to use. Only used for binary problems with objectives whose thresholds can be tuned.

**Parameters**

- **X** (*pd.DataFrame*) – Input features.
- **y** (*pd.Series*) – Input target values.
- **y_pred_proba** (*pd.Series*) – The predicted probabilities of the target outputted by the pipeline.

- **objective** (*ObjectiveBase*) – The objective to threshold with. Must have a tunable threshold.

**Raises** **ValueError** – If objective is not optimizable.

**property parameters**(*self*)

Parameter dictionary for this pipeline.

**Returns** Dictionary of all component parameters.

**Return type** dict

**predict**(*self*, *X*, *objective=None*, *X_train=None*, *y_train=None*)

Predict on future data where target is not known.

**Parameters**

- **X** (*pd.DataFrame, or np.ndarray*) – Data of shape [n_samples, n_features].
- **objective** (*Object or string*) – The objective to use to make predictions.
- **X_train** (*pd.DataFrame or np.ndarray or None*) – Training data.
- **y_train** (*pd.Series or None*) – Training labels.

**Raises** **ValueError** – If X_train and/or y_train are None or if final component is not an Estimator.

**Returns** Predictions.

**predict_in_sample**(*self*, *X*, *y*, *X_train*, *y_train*, *objective=None*)

Predict on future data where the target is known, e.g. cross validation.

**Parameters**

- **X** (*pd.DataFrame*) – Future data of shape [n_samples, n_features].
- **y** (*pd.Series*) – Future target of shape [n_samples].
- **X_train** (*pd.DataFrame*) – Data the pipeline was trained on of shape [n_samples_train, n_feautures].
- **y_train** (*pd.Series*) – Targets used to train the pipeline of shape [n_samples_train].
- **objective** (*ObjectiveBase, str*) – Objective used to threshold predicted probabilities, optional. Defaults to None.

**Returns** Estimated labels.

**Return type** pd.Series

**Raises** **ValueError** – If objective is not defined for time-series binary classification problems.

**predict_proba**(*self*, *X*, *X_train=None*, *y_train=None*)

Predict on future data where the target is unknown.

**Parameters**

- **X** (*pd.DataFrame or np.ndarray*) – Future data of shape [n_samples, n_features].
- **X_train** (*pd.DataFrame, np.ndarray*) – Data the pipeline was trained on of shape [n_samples_train, n_features].
- **y_train** (*pd.Series, np.ndarray*) – Targets used to train the pipeline of shape [n_samples_train].

**Returns** Estimated probabilities.

**Return type** pd.Series

**Raises** `ValueError` – If final component is not an Estimator.

`predict_proba_in_sample`(*self*, *X_holdout*, *y_holdout*, *X_train*, *y_train*)

Predict on future data where the target is known, e.g. cross validation.

**Parameters**

- `X_holdout` (*pd.DataFrame or np.ndarray*) – Future data of shape [n_samples, n_features].
- `y_holdout` (*pd.Series, np.ndarray*) – Future target of shape [n_samples].
- `X_train` (*pd.DataFrame, np.ndarray*) – Data the pipeline was trained on of shape [n_samples_train, n_features].
- `y_train` (*pd.Series, np.ndarray*) – Targets used to train the pipeline of shape [n_samples_train].

**Returns** Estimated probabilities.

**Return type** pd.Series

**Raises** `ValueError` – If the final component is not an Estimator.

`save`(*self*, *file_path*, *pickle_protocol=cloudpickle.DEFAULT_PROTOCOL*)

Saves pipeline at file path.

**Parameters**

- `file_path` (*str*) – Location to save file.
- `pickle_protocol` (*int*) – The pickle data stream format.

`score`(*self*, *X*, *y*, *objectives*, *X_train=None*, *y_train=None*)

Evaluate model performance on current and additional objectives.

**Parameters**

- `X` (*pd.DataFrame or np.ndarray*) – Data of shape [n_samples, n_features].
- `y` (*pd.Series*) – True labels of length [n_samples].
- `objectives` (*list*) – Non-empty list of objectives to score on.
- `X_train` (*pd.DataFrame, np.ndarray*) – Data the pipeline was trained on of shape [n_samples_train, n_features].
- `y_train` (*pd.Series, np.ndarray*) – Targets used to train the pipeline of shape [n_samples_train].

**Returns** Ordered dictionary of objective scores.

**Return type** dict

`property summary`(*self*)

A short summary of the pipeline structure, describing the list of components used.

Example: Logistic Regression Classifier w/ Simple Imputer + One Hot Encoder

**Returns** A string describing the pipeline structure.

`property threshold`(*self*)

Threshold used to make a prediction. Defaults to None.

**transform**(*self*, *X*, *y=None*)

Transform the input.

**Parameters**

- **X** (*pd.DataFrame, or np.ndarray*) – Data of shape [n_samples, n_features].
- **y** (*pd.Series*) – The target data of length [n_samples]. Defaults to None.

**Returns** Transformed output.

**Return type** pd.DataFrame

**transform_all_but_final**(*self*, *X*, *y=None*, *X_train=None*, *y_train=None*, *calculating_residuals=False*)

Transforms the data by applying all pre-processing components.

**Parameters**

- **X** (*pd.DataFrame*) – Input data to the pipeline to transform.
- **y** (*pd.Series*) – Targets corresponding to the pipeline targets.
- **X_train** (*pd.DataFrame*) – Training data used to generate generates from past observations.
- **y_train** (*pd.Series*) – Training targets used to generate features from past observations.
- **calculating_residuals** (*bool*) – Whether we're calling predict_in_sample to calculate the residuals. This means the X and y arguments are not future data, but actually the train data.

**Returns** New transformed features.

**Return type** pd.DataFrame

**class** evalml.pipelines.**TimeSeriesClassificationPipeline**(*component_graph*, *parameters=None*, *custom_name=None*, *random_seed=0*)

Pipeline base class for time series classification problems.

**Parameters**

- **component_graph** (ComponentGraph, *list, dict*) – ComponentGraph instance, list of components in order, or dictionary of components. Accepts strings or ComponentBase subclasses in the list. Note that when duplicate components are specified in a list, the duplicate component names will be modified with the component's index in the list. For example, the component graph [Imputer, One Hot Encoder, Imputer, Logistic Regression Classifier] will have names ["Imputer", "One Hot Encoder", "Imputer_2", "Logistic Regression Classifier"]
- **parameters** (*dict*) – Dictionary with component names as keys and dictionary of that component's parameters as values. An empty dictionary { } implies using all default values for component parameters. Pipeline-level parameters such as time_index, gap, and max_delay must be specified with the "pipeline" key. For example: Pipeline(parameters={"pipeline": {"time_index": "Date", "max_delay": 4, "gap": 2}}).
- **random_seed** (*int*) – Seed for the random number generator. Defaults to 0.

**Attributes**

| | |
|---|---|
| **problem_type** | None |

**Methods**

| | |
|---|---|
| *can_tune_threshold_with_objective* | Determine whether the threshold of a binary classification pipeline can be tuned. |
| *classes_* | Gets the class names for the pipeline. Will return None before pipeline is fit. |
| *clone* | Constructs a new pipeline with the same components, parameters, and random seed. |
| *create_objectives* | Create objective instances from a list of strings or objective classes. |
| *custom_name* | Custom name of the pipeline. |
| *dates_needed_for_prediction* | Return dates needed to forecast the given date in the future. |
| *dates_needed_for_prediction_range* | Return dates needed to forecast the given date in the future. |
| *describe* | Outputs pipeline details including component parameters. |
| *feature_importance* | Importance associated with each feature. Features dropped by the feature selection are excluded. |
| *fit* | Fit a time series classification model. |
| *fit_transform* | Fit and transform all components in the component graph, if all components are Transformers. |
| *get_component* | Returns component by name. |
| *get_hyperparameter_ranges* | Returns hyperparameter ranges from all components as a dictionary. |
| *graph* | Generate an image representing the pipeline graph. |
| *graph_dict* | Generates a dictionary with nodes consisting of the component names and parameters, and edges detailing component relationships. This dictionary is JSON serializable in most cases. |
| *graph_feature_importance* | Generate a bar graph of the pipeline's feature importance. |
| *inverse_transform* | Apply component inverse_transform methods to estimator predictions in reverse order. |
| *load* | Loads pipeline at file path. |
| *model_family* | Returns model family of this pipeline. |
| *name* | Name of the pipeline. |
| *new* | Constructs a new instance of the pipeline with the same component graph but with a different set of parameters. Not to be confused with python's __new__ method. |
| *parameters* | Parameter dictionary for this pipeline. |
| *predict* | Predict on future data where target is not known. |
| *predict_in_sample* | Predict on future data where the target is known, e.g. cross validation. |
| *predict_proba* | Predict on future data where the target is unknown. |
| *predict_proba_in_sample* | Predict on future data where the target is known, e.g. cross validation. |
| *save* | Saves pipeline at file path. |
| *score* | Evaluate model performance on current and additional objectives. |
| *summary* | A short summary of the pipeline structure, describing the list of components used. |

continues on next page

Table 13 – continued from previous page

| | |
|---|---|
| *transform* | Transform the input. |
| *transform_all_but_final* | Transforms the data by applying all pre-processing components. |

**can_tune_threshold_with_objective**(*self*, *objective*)

Determine whether the threshold of a binary classification pipeline can be tuned.

**Parameters** **objective** (*ObjectiveBase*) – Primary AutoMLSearch objective.

**Returns** True if the pipeline threshold can be tuned.

**Return type** bool

**property classes_**(*self*)

Gets the class names for the pipeline. Will return None before pipeline is fit.

**clone**(*self*)

Constructs a new pipeline with the same components, parameters, and random seed.

**Returns** A new instance of this pipeline with identical components, parameters, and random seed.

**static create_objectives**(*objectives*)

Create objective instances from a list of strings or objective classes.

**property custom_name**(*self*)

Custom name of the pipeline.

**dates_needed_for_prediction**(*self*, *date*)

Return dates needed to forecast the given date in the future.

**Parameters** **date** (*pd.Timestamp*) – Date to forecast in the future.

**Returns** Range of dates needed to forecast the given date.

**Return type** dates_needed (tuple(pd.Timestamp))

**dates_needed_for_prediction_range**(*self*, *start_date*, *end_date*)

Return dates needed to forecast the given date in the future.

**Parameters**

- **start_date** (*pd.Timestamp*) – Start date of range to forecast in the future.
- **end_date** (*pd.Timestamp*) – End date of range to forecast in the future.

**Returns** Range of dates needed to forecast the given date.

**Return type** dates_needed (tuple(pd.Timestamp))

**Raises** **ValueError** – If start_date doesn't come before end_date

**describe**(*self*, *return_dict=False*)

Outputs pipeline details including component parameters.

**Parameters** **return_dict** (*bool*) – If True, return dictionary of information about pipeline. Defaults to False.

**Returns** Dictionary of all component parameters if return_dict is True, else None.

**Return type** dict

**property feature_importance**(*self*)

Importance associated with each feature. Features dropped by the feature selection are excluded.

**Returns** Feature names and their corresponding importance

**Return type** pd.DataFrame

**fit**(*self*, *X*, *y*)

Fit a time series classification model.

**Parameters**

- **X** (*pd.DataFrame or np.ndarray*) – The input training data of shape [n_samples, n_features]
- **y** (*pd.Series, np.ndarray*) – The target training labels of length [n_samples]

**Returns** self

**Raises** **ValueError** – If the number of unique classes in y are not appropriate for the type of pipeline.

**fit_transform**(*self*, *X*, *y*)

Fit and transform all components in the component graph, if all components are Transformers.

**Parameters**

- **X** (*pd.DataFrame*) – Input features of shape [n_samples, n_features].
- **y** (*pd.Series*) – The target data of length [n_samples].

**Returns** Transformed output.

**Return type** pd.DataFrame

**Raises** **ValueError** – If final component is an Estimator.

**get_component**(*self*, *name*)

Returns component by name.

**Parameters** **name** (*str*) – Name of component.

**Returns** Component to return

**Return type** Component

**get_hyperparameter_ranges**(*self*, *custom_hyperparameters*)

Returns hyperparameter ranges from all components as a dictionary.

**Parameters** **custom_hyperparameters** (*dict*) – Custom hyperparameters for the pipeline.

**Returns** Dictionary of hyperparameter ranges for each component in the pipeline.

**Return type** dict

**graph**(*self*, *filepath=None*)

Generate an image representing the pipeline graph.

**Parameters** **filepath** (*str, optional*) – Path to where the graph should be saved. If set to None (as by default), the graph will not be saved.

**Returns** Graph object that can be directly displayed in Jupyter notebooks.

**Return type** graphviz.Digraph

**Raises**

- **RuntimeError** – If graphviz is not installed.
- **ValueError** – If path is not writeable.

**graph_dict**(*self*)

Generates a dictionary with nodes consisting of the component names and parameters, and edges detailing component relationships. This dictionary is JSON serializable in most cases.

x_edges specifies from which component feature data is being passed. y_edges specifies from which component target data is being passed. This can be used to build graphs across a variety of visualization tools. Template: {"Nodes": {"component_name": {"Name": class_name, "Parameters": parameters_attributes}, ...}}, "x_edges": [[from_component_name, to_component_name], [from_component_name, to_component_name], ...], "y_edges": [[from_component_name, to_component_name], [from_component_name, to_component_name], ...]}

**Returns** A dictionary representing the DAG structure.

**Return type** dag_dict (dict)

**graph_feature_importance**(*self*, *importance_threshold=0*)

Generate a bar graph of the pipeline's feature importance.

**Parameters** **importance_threshold** (*float, optional*) – If provided, graph features with a permutation importance whose absolute value is larger than importance_threshold. Defaults to zero.

**Returns** A bar graph showing features and their corresponding importance.

**Return type** plotly.Figure

**Raises** **ValueError** – If importance threshold is not valid.

**inverse_transform**(*self*, *y*)

Apply component inverse_transform methods to estimator predictions in reverse order.

Components that implement inverse_transform are PolynomialDecomposer, LogTransformer, LabelEncoder (tbd).

**Parameters** **y** (*pd.Series*) – Final component features.

**Returns** The inverse transform of the target.

**Return type** pd.Series

**static load**(*file_path*)

Loads pipeline at file path.

**Parameters** **file_path** (*str*) – Location to load file.

**Returns** PipelineBase object

**property model_family**(*self*)

Returns model family of this pipeline.

**property name**(*self*)

Name of the pipeline.

**new**(*self*, *parameters*, *random_seed=0*)

Constructs a new instance of the pipeline with the same component graph but with a different set of parameters. Not to be confused with python's __new__ method.

**Parameters**

- **parameters** (*dict*) – Dictionary with component names as keys and dictionary of that component's parameters as values. An empty dictionary or None implies using all default values for component parameters. Defaults to None.
- **random_seed** (*int*) – Seed for the random number generator. Defaults to 0.

**Returns** A new instance of this pipeline with identical components.

**property parameters**(*self*)

Parameter dictionary for this pipeline.

**Returns** Dictionary of all component parameters.

**Return type** dict

**predict**(*self*, *X*, *objective=None*, *X_train=None*, *y_train=None*)

Predict on future data where target is not known.

**Parameters**

- **X** (*pd.DataFrame, or np.ndarray*) – Data of shape [n_samples, n_features].
- **objective** (*Object or string*) – The objective to use to make predictions.
- **X_train** (*pd.DataFrame or np.ndarray or None*) – Training data.
- **y_train** (*pd.Series or None*) – Training labels.

**Raises** **ValueError** – If X_train and/or y_train are None or if final component is not an Estimator.

**Returns** Predictions.

**predict_in_sample**(*self*, *X*, *y*, *X_train*, *y_train*, *objective=None*)

Predict on future data where the target is known, e.g. cross validation.

Note: we cast y as ints first to address boolean values that may be returned from calculating predictions which we would not be able to otherwise transform if we originally had integer targets.

**Parameters**

- **X** (*pd.DataFrame or np.ndarray*) – Future data of shape [n_samples, n_features].
- **y** (*pd.Series, np.ndarray*) – Future target of shape [n_samples].
- **X_train** (*pd.DataFrame, np.ndarray*) – Data the pipeline was trained on of shape [n_samples_train, n_features].
- **y_train** (*pd.Series, np.ndarray*) – Targets used to train the pipeline of shape [n_samples_train].
- **objective** (*ObjectiveBase, str, None*) – Objective used to threshold predicted probabilities, optional.

**Returns** Estimated labels.

**Return type** pd.Series

**Raises** **ValueError** – If final component is not an Estimator.

**predict_proba**(*self*, *X*, *X_train=None*, *y_train=None*)

Predict on future data where the target is unknown.

**Parameters**

- **X** (*pd.DataFrame or np.ndarray*) – Future data of shape [n_samples, n_features].

- **X_train** (*pd.DataFrame, np.ndarray*) – Data the pipeline was trained on of shape [n_samples_train, n_features].
- **y_train** (*pd.Series, np.ndarray*) – Targets used to train the pipeline of shape [n_samples_train].

**Returns** Estimated probabilities.

**Return type** pd.Series

**Raises** **ValueError** – If final component is not an Estimator.

**predict_proba_in_sample**(*self*, *X_holdout*, *y_holdout*, *X_train*, *y_train*)

Predict on future data where the target is known, e.g. cross validation.

**Parameters**

- **X_holdout** (*pd.DataFrame or np.ndarray*) – Future data of shape [n_samples, n_features].
- **y_holdout** (*pd.Series, np.ndarray*) – Future target of shape [n_samples].
- **X_train** (*pd.DataFrame, np.ndarray*) – Data the pipeline was trained on of shape [n_samples_train, n_features].
- **y_train** (*pd.Series, np.ndarray*) – Targets used to train the pipeline of shape [n_samples_train].

**Returns** Estimated probabilities.

**Return type** pd.Series

**Raises** **ValueError** – If the final component is not an Estimator.

**save**(*self*, *file_path*, *pickle_protocol=cloudpickle.DEFAULT_PROTOCOL*)

Saves pipeline at file path.

**Parameters**

- **file_path** (*str*) – Location to save file.
- **pickle_protocol** (*int*) – The pickle data stream format.

**score**(*self*, *X*, *y*, *objectives*, *X_train=None*, *y_train=None*)

Evaluate model performance on current and additional objectives.

**Parameters**

- **X** (*pd.DataFrame or np.ndarray*) – Data of shape [n_samples, n_features].
- **y** (*pd.Series*) – True labels of length [n_samples].
- **objectives** (*list*) – Non-empty list of objectives to score on.
- **X_train** (*pd.DataFrame, np.ndarray*) – Data the pipeline was trained on of shape [n_samples_train, n_features].
- **y_train** (*pd.Series, np.ndarray*) – Targets used to train the pipeline of shape [n_samples_train].

**Returns** Ordered dictionary of objective scores.

**Return type** dict

**property summary**(*self*)

A short summary of the pipeline structure, describing the list of components used.

Example: Logistic Regression Classifier w/ Simple Imputer + One Hot Encoder

**Returns** A string describing the pipeline structure.

**transform**(*self*, *X*, *y=None*)

Transform the input.

**Parameters**

- **X** (*pd.DataFrame, or np.ndarray*) – Data of shape [n_samples, n_features].
- **y** (*pd.Series*) – The target data of length [n_samples]. Defaults to None.

**Returns** Transformed output.

**Return type** pd.DataFrame

**transform_all_but_final**(*self*, *X*, *y=None*, *X_train=None*, *y_train=None*, *calculating_residuals=False*)

Transforms the data by applying all pre-processing components.

**Parameters**

- **X** (*pd.DataFrame*) – Input data to the pipeline to transform.
- **y** (*pd.Series*) – Targets corresponding to the pipeline targets.
- **X_train** (*pd.DataFrame*) – Training data used to generate generates from past observations.
- **y_train** (*pd.Series*) – Training targets used to generate features from past observations.
- **calculating_residuals** (*bool*) – Whether we're calling predict_in_sample to calculate the residuals. This means the X and y arguments are not future data, but actually the train data.

**Returns** New transformed features.

**Return type** pd.DataFrame

**class** evalml.pipelines.**TimeSeriesFeaturizer**(*time_index=None*, *max_delay=2*, *gap=0*, *forecast_horizon=1*, *conf_level=0.05*, *rolling_window_size=0.25*, *delay_features=True*, *delay_target=True*, *random_seed=0*, ***kwargs*)

Transformer that delays input features and target variable for time series problems.

This component uses an algorithm based on the autocorrelation values of the target variable to determine which lags to select from the set of all possible lags.

The algorithm is based on the idea that the local maxima of the autocorrelation function indicate the lags that have the most impact on the present time.

The algorithm computes the autocorrelation values and finds the local maxima, called "peaks", that are significant at the given conf_level. Since lags in the range [0, 10] tend to be predictive but not local maxima, the union of the peaks is taken with the significant lags in the range [0, 10]. At the end, only selected lags in the range [0, max_delay] are used.

Parametrizing the algorithm by conf_level lets the AutoMLAlgorithm tune the set of lags chosen so that the chances of finding a good set of lags is higher.

Using conf_level value of 1 selects all possible lags.

**Parameters**

- **time_index** (*str*) – Name of the column containing the datetime information used to order the data. Ignored.
- **max_delay** (*int*) – Maximum number of time units to delay each feature. Defaults to 2.
- **forecast_horizon** (*int*) – The number of time periods the pipeline is expected to forecast.
- **conf_level** (*float*) – Float in range (0, 1] that determines the confidence interval size used to select which lags to compute from the set of [1, max_delay]. A delay of 1 will always be computed. If 1, selects all possible lags in the set of [1, max_delay], inclusive.
- **rolling_window_size** (*float*) – Float in range (0, 1] that determines the size of the window used for rolling features. Size is computed as rolling_window_size * max_delay.
- **delay_features** (*bool*) – Whether to delay the input features. Defaults to True.
- **delay_target** (*bool*) – Whether to delay the target. Defaults to True.
- **gap** (*int*) – The number of time units between when the features are collected and when the target is collected. For example, if you are predicting the next time step's target, gap=1. This is only needed because when gap=0, we need to be sure to start the lagging of the target variable at 1. Defaults to 1.
- **random_seed** (*int*) – Seed for the random number generator. This transformer performs the same regardless of the random seed provided.

**Attributes**

| | |
|---|---|
| **hyper-parameter_ranges** | Real(0.001, 1.0), "rolling_window_size": Real(0.001, 1.0)}:type: {"conf_level" |
| **modifies_features** | True |
| **modifies_target** | False |
| **name** | Time Series Featurizer |
| **needs_fitting** | True |
| **target_colname_prefix** | target_delay_{} |
| **training_only** | False |

**Methods**

| | |
|---|---|
| *clone* | Constructs a new component with the same parameters and random state. |
| *default_parameters* | Returns the default parameters for this component. |
| *describe* | Describe a component and its parameters. |
| *fit* | Fits the DelayFeatureTransformer. |
| *fit_transform* | Fit the component and transform the input data. |
| *load* | Loads component at file path. |
| *parameters* | Returns the parameters which were used to initialize the component. |
| *save* | Saves component at file path. |
| *transform* | Computes the delayed values and rolling means for X and y. |
| *update_parameters* | Updates the parameter dictionary of the component. |

**clone**(*self*)

Constructs a new component with the same parameters and random state.

**Returns** A new instance of this component with identical parameters and random state.

**default_parameters**(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

**Returns** Default parameters for this component.

**Return type** dict

**describe**(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

**Parameters**

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description as dictionary in the format {"name": name, "parameters": parameters}

**Returns** Returns dictionary if return_dict is True, else None.

**Return type** None or dict

**fit**(*self*, *X*, *y=None*)

Fits the DelayFeatureTransformer.

**Parameters**

- **X** (*pd.DataFrame or np.ndarray*) – The input training data of shape [n_samples, n_features]
- **y** (*pd.Series, optional*) – The target training data of length [n_samples]

**Returns** self

**Raises** **ValueError** – if self.time_index is None

**fit_transform**(*self*, *X*, *y=None*)

Fit the component and transform the input data.

**Parameters**

- **X** (*pd.DataFrame*) – Data to transform.
- **y** (*pd.Series, or None*) – Target.

**Returns** Transformed X.

**Return type** pd.DataFrame

**static load**(*file_path*)

Loads component at file path.

**Parameters** **file_path** (*str*) – Location to load file.

**Returns** ComponentBase object

**property parameters**(*self*)

Returns the parameters which were used to initialize the component.

**save**(*self*, *file_path*, *pickle_protocol=cloudpickle.DEFAULT_PROTOCOL*)

Saves component at file path.

**Parameters**

- **file_path** (*str*) – Location to save file.
- **pickle_protocol** (*int*) – The pickle data stream format.

**transform**(*self*, *X*, *y=None*)

Computes the delayed values and rolling means for X and y.

The chosen delays are determined by the autocorrelation function of the target variable. See the class docstring for more information on how they are chosen. If y is None, all possible lags are chosen.

If y is not None, it will also compute the delayed values for the target variable.

The rolling means for all numeric features in X and y, if y is numeric, are also returned.

**Parameters**

- **X** (*pd.DataFrame or None*) – Data to transform. None is expected when only the target variable is being used.
- **y** (*pd.Series, or None*) – Target.

**Returns** Transformed X. No original features are returned.

**Return type** pd.DataFrame

**update_parameters**(*self*, *update_dict*, *reset_fit=True*)

Updates the parameter dictionary of the component.

**Parameters**

- **update_dict** (*dict*) – A dict of parameters to update.
- **reset_fit** (*bool, optional*) – If True, will set *_is_fitted* to False.

**class** evalml.pipelines.**TimeSeriesImputer**(*categorical_impute_strategy='forwards_fill'*, *numeric_impute_strategy='interpolate'*, *target_impute_strategy='forwards_fill'*, *random_seed=0*, ***kwargs*)

Imputes missing data according to a specified timeseries-specific imputation strategy.

This Transformer should be used after the *TimeSeriesRegularizer* in order to impute the missing values that were added to X and y (if passed).

**Parameters**

- **categorical_impute_strategy** (*string*) – Impute strategy to use for string, object, boolean, categorical dtypes. Valid values include "backwards_fill" and "forwards_fill". Defaults to "forwards_fill".
- **numeric_impute_strategy** (*string*) – Impute strategy to use for numeric columns. Valid values include "backwards_fill", "forwards_fill", and "interpolate". Defaults to "interpolate".
- **target_impute_strategy** (*string*) – Impute strategy to use for the target column. Valid values include "backwards_fill", "forwards_fill", and "interpolate". Defaults to "forwards_fill".
- **random_seed** (*int*) – Seed for the random number generator. Defaults to 0.

**Raises** `ValueError` – If categorical_impute_strategy, numeric_impute_strategy, or target_impute_strategy is not one of the valid values.

**Attributes**

| | |
|---|---|
| **hyperparameter_ranges** | { "categorical_impute_strategy": ["backwards_fill", "forwards_fill"], "numeric_impute_strategy": ["backwards_fill", "forwards_fill", "interpolate"], "target_impute_strategy": ["backwards_fill", "forwards_fill", "interpolate"],} |
| **modifies_features** | True |
| **modifies_target** | True |
| **name** | Time Series Imputer |
| **training_only** | True |

**Methods**

| | |
|---|---|
| *clone* | Constructs a new component with the same parameters and random state. |
| *default_parameters* | Returns the default parameters for this component. |
| *describe* | Describe a component and its parameters. |
| *fit* | Fits imputer to data. |
| *fit_transform* | Fits on X and transforms X. |
| *load* | Loads component at file path. |
| *needs_fitting* | Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances. |
| *parameters* | Returns the parameters which were used to initialize the component. |
| *save* | Saves component at file path. |
| *transform* | Transforms data X by imputing missing values using specified timeseries-specific strategies. 'None' values are converted to np.nan before imputation and are treated as the same. |
| *update_parameters* | Updates the parameter dictionary of the component. |

`clone`(*self*)

Constructs a new component with the same parameters and random state.

**Returns** A new instance of this component with identical parameters and random state.

`default_parameters`(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

**Returns** Default parameters for this component.

**Return type** dict

`describe`(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

**Parameters**

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description as dictionary in the format {"name": name, "parameters": parameters}

**Returns** Returns dictionary if return_dict is True, else None.

**Return type** None or dict

**fit**(*self*, *X*, *y=None*)

Fits imputer to data.

'None' values are converted to np.nan before imputation and are treated as the same. If a value is missing at the beginning or end of a column, that value will be imputed using backwards fill or forwards fill as necessary, respectively.

**Parameters**

- **X** (*pd.DataFrame, np.ndarray*) – The input training data of shape [n_samples, n_features]
- **y** (*pd.Series, optional*) – The target training data of length [n_samples]

**Returns** self

**fit_transform**(*self*, *X*, *y=None*)

Fits on X and transforms X.

**Parameters**

- **X** (*pd.DataFrame*) – Data to fit and transform.
- **y** (*pd.Series*) – Target data.

**Returns** Transformed X.

**Return type** pd.DataFrame

**Raises** **MethodPropertyNotFoundError** – If transformer does not have a transform method or a component_obj that implements transform.

**static load**(*file_path*)

Loads component at file path.

**Parameters** **file_path** (*str*) – Location to load file.

**Returns** ComponentBase object

**needs_fitting**(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need to be fit or whose fit methods do nothing.

**Returns** True.

**property parameters**(*self*)

Returns the parameters which were used to initialize the component.

**save**(*self*, *file_path*, *pickle_protocol=cloudpickle.DEFAULT_PROTOCOL*)

Saves component at file path.

**Parameters**

- **file_path** (*str*) – Location to save file.

- **pickle_protocol** (*int*) – The pickle data stream format.

**transform**(*self*, *X*, *y=None*)

Transforms data X by imputing missing values using specified timeseries-specific strategies. 'None' values are converted to np.nan before imputation and are treated as the same.

**Parameters**

- **X** (*pd.DataFrame*) – Data to transform.
- **y** (*pd.Series, optional*) – Optionally, target data to transform.

**Returns** Transformed X and y

**Return type** pd.DataFrame

**update_parameters**(*self*, *update_dict*, *reset_fit=True*)

Updates the parameter dictionary of the component.

**Parameters**

- **update_dict** (*dict*) – A dict of parameters to update.
- **reset_fit** (*bool, optional*) – If True, will set *_is_fitted* to False.

**class** evalml.pipelines.**TimeSeriesMulticlassClassificationPipeline**(*component_graph*, *parameters=None*, *custom_name=None*, *random_seed=0*)

Pipeline base class for time series multiclass classification problems.

**Parameters**

- **component_graph** (*list or dict*) – List of components in order. Accepts strings or ComponentBase subclasses in the list. Note that when duplicate components are specified in a list, the duplicate component names will be modified with the component's index in the list. For example, the component graph [Imputer, One Hot Encoder, Imputer, Logistic Regression Classifier] will have names ["Imputer", "One Hot Encoder", "Imputer_2", "Logistic Regression Classifier"]
- **parameters** (*dict*) – Dictionary with component names as keys and dictionary of that component's parameters as values. An empty dictionary {} implies using all default values for component parameters. Pipeline-level parameters such as time_index, gap, and max_delay must be specified with the "pipeline" key. For example: Pipeline(parameters={"pipeline": {"time_index": "Date", "max_delay": 4, "gap": 2}}).
- **random_seed** (*int*) – Seed for the random number generator. Defaults to 0.

### Example

```
>>> pipeline = TimeSeriesMulticlassClassificationPipeline(component_graph=["Simple␣
↪Imputer", "Logistic Regression Classifier"],
...                                                        parameters={"Logistic␣
↪Regression Classifier": {"penalty": "elasticnet",
...                                                                                ␣
↪                        "solver": "liblinear"},
...                                                                  "pipeline": {
↪"gap": 1, "max_delay": 1, "forecast_horizon": 1, "time_index": "date"}},
...                                                        custom_name="My␣
↪TimeSeriesMulticlass Pipeline")
```

(continues on next page)

(continued from previous page)

```
>>> assert pipeline.custom_name == "My TimeSeriesMulticlass Pipeline"
>>> assert pipeline.component_graph.component_dict.keys() == {'Simple Imputer',
↪'Logistic Regression Classifier'}
>>> assert pipeline.parameters == {
...  'Simple Imputer': {'impute_strategy': 'most_frequent', 'fill_value': None},
...  'Logistic Regression Classifier': {'penalty': 'elasticnet',
...                                     'C': 1.0,
...                                     'n_jobs': -1,
...                                     'multi_class': 'auto',
...                                     'solver': 'liblinear'},
...     'pipeline': {'gap': 1, 'max_delay': 1, 'forecast_horizon': 1, 'time_index':
↪"date"}}
```

**Attributes**

| | |
|---|---|
| **problem_type** | ProblemTypes.TIME_SERIES_MULTICLASS |

**Methods**

| | |
|---|---|
| *can_tune_threshold_with_objective* | Determine whether the threshold of a binary classification pipeline can be tuned. |
| *classes_* | Gets the class names for the pipeline. Will return None before pipeline is fit. |
| *clone* | Constructs a new pipeline with the same components, parameters, and random seed. |
| *create_objectives* | Create objective instances from a list of strings or objective classes. |
| *custom_name* | Custom name of the pipeline. |
| *dates_needed_for_prediction* | Return dates needed to forecast the given date in the future. |
| *dates_needed_for_prediction_range* | Return dates needed to forecast the given date in the future. |
| *describe* | Outputs pipeline details including component parameters. |
| *feature_importance* | Importance associated with each feature. Features dropped by the feature selection are excluded. |
| *fit* | Fit a time series classification model. |
| *fit_transform* | Fit and transform all components in the component graph, if all components are Transformers. |
| *get_component* | Returns component by name. |
| *get_hyperparameter_ranges* | Returns hyperparameter ranges from all components as a dictionary. |
| *graph* | Generate an image representing the pipeline graph. |
| *graph_dict* | Generates a dictionary with nodes consisting of the component names and parameters, and edges detailing component relationships. This dictionary is JSON serializable in most cases. |
| *graph_feature_importance* | Generate a bar graph of the pipeline's feature importance. |

continues on next page

Table 14 – continued from previous page

| | |
|---|---|
| *inverse_transform* | Apply component inverse_transform methods to estimator predictions in reverse order. |
| *load* | Loads pipeline at file path. |
| *model_family* | Returns model family of this pipeline. |
| *name* | Name of the pipeline. |
| *new* | Constructs a new instance of the pipeline with the same component graph but with a different set of parameters. Not to be confused with python's __new__ method. |
| *parameters* | Parameter dictionary for this pipeline. |
| *predict* | Predict on future data where target is not known. |
| *predict_in_sample* | Predict on future data where the target is known, e.g. cross validation. |
| *predict_proba* | Predict on future data where the target is unknown. |
| *predict_proba_in_sample* | Predict on future data where the target is known, e.g. cross validation. |
| *save* | Saves pipeline at file path. |
| *score* | Evaluate model performance on current and additional objectives. |
| *summary* | A short summary of the pipeline structure, describing the list of components used. |
| *transform* | Transform the input. |
| *transform_all_but_final* | Transforms the data by applying all pre-processing components. |

**can_tune_threshold_with_objective**(*self*, *objective*)

Determine whether the threshold of a binary classification pipeline can be tuned.

**Parameters** **objective** (*ObjectiveBase*) – Primary AutoMLSearch objective.

**Returns** True if the pipeline threshold can be tuned.

**Return type** bool

**property classes_**(*self*)

Gets the class names for the pipeline. Will return None before pipeline is fit.

**clone**(*self*)

Constructs a new pipeline with the same components, parameters, and random seed.

**Returns** A new instance of this pipeline with identical components, parameters, and random seed.

**static create_objectives**(*objectives*)

Create objective instances from a list of strings or objective classes.

**property custom_name**(*self*)

Custom name of the pipeline.

**dates_needed_for_prediction**(*self*, *date*)

Return dates needed to forecast the given date in the future.

**Parameters** **date** (*pd.Timestamp*) – Date to forecast in the future.

**Returns** Range of dates needed to forecast the given date.

**Return type** dates_needed (tuple(pd.Timestamp))

**dates_needed_for_prediction_range**(*self*, *start_date*, *end_date*)

Return dates needed to forecast the given date in the future.

**Parameters**

- **start_date** (*pd.Timestamp*) – Start date of range to forecast in the future.
- **end_date** (*pd.Timestamp*) – End date of range to forecast in the future.

**Returns** Range of dates needed to forecast the given date.

**Return type** dates_needed (tuple(pd.Timestamp))

**Raises** **ValueError** – If start_date doesn't come before end_date

**describe**(*self*, *return_dict=False*)

Outputs pipeline details including component parameters.

**Parameters** **return_dict** (*bool*) – If True, return dictionary of information about pipeline. Defaults to False.

**Returns** Dictionary of all component parameters if return_dict is True, else None.

**Return type** dict

**property feature_importance**(*self*)

Importance associated with each feature. Features dropped by the feature selection are excluded.

**Returns** Feature names and their corresponding importance

**Return type** pd.DataFrame

**fit**(*self*, *X*, *y*)

Fit a time series classification model.

**Parameters**

- **X** (*pd.DataFrame or np.ndarray*) – The input training data of shape [n_samples, n_features]
- **y** (*pd.Series, np.ndarray*) – The target training labels of length [n_samples]

**Returns** self

**Raises** **ValueError** – If the number of unique classes in y are not appropriate for the type of pipeline.

**fit_transform**(*self*, *X*, *y*)

Fit and transform all components in the component graph, if all components are Transformers.

**Parameters**

- **X** (*pd.DataFrame*) – Input features of shape [n_samples, n_features].
- **y** (*pd.Series*) – The target data of length [n_samples].

**Returns** Transformed output.

**Return type** pd.DataFrame

**Raises** **ValueError** – If final component is an Estimator.

**get_component**(*self*, *name*)

Returns component by name.

**Parameters** **name** (*str*) – Name of component.

**Returns** Component to return

**Return type** Component

`get_hyperparameter_ranges`(*self*, *custom_hyperparameters*)

Returns hyperparameter ranges from all components as a dictionary.

**Parameters** `custom_hyperparameters` (*dict*) – Custom hyperparameters for the pipeline.

**Returns** Dictionary of hyperparameter ranges for each component in the pipeline.

**Return type** dict

`graph`(*self*, *filepath=None*)

Generate an image representing the pipeline graph.

**Parameters** `filepath` (*str, optional*) – Path to where the graph should be saved. If set to None (as by default), the graph will not be saved.

**Returns** Graph object that can be directly displayed in Jupyter notebooks.

**Return type** graphviz.Digraph

**Raises**

- `RuntimeError` – If graphviz is not installed.
- `ValueError` – If path is not writeable.

`graph_dict`(*self*)

Generates a dictionary with nodes consisting of the component names and parameters, and edges detailing component relationships. This dictionary is JSON serializable in most cases.

x_edges specifies from which component feature data is being passed. y_edges specifies from which component target data is being passed. This can be used to build graphs across a variety of visualization tools. Template: {"Nodes": {"component_name": {"Name": class_name, "Parameters": parameters_attributes}, ...}}, "x_edges": [[from_component_name, to_component_name], [from_component_name, to_component_name], ...], "y_edges": [[from_component_name, to_component_name], [from_component_name, to_component_name], ...]}

**Returns** A dictionary representing the DAG structure.

**Return type** dag_dict (dict)

`graph_feature_importance`(*self*, *importance_threshold=0*)

Generate a bar graph of the pipeline's feature importance.

**Parameters** `importance_threshold` (*float, optional*) – If provided, graph features with a permutation importance whose absolute value is larger than importance_threshold. Defaults to zero.

**Returns** A bar graph showing features and their corresponding importance.

**Return type** plotly.Figure

**Raises** `ValueError` – If importance threshold is not valid.

`inverse_transform`(*self*, *y*)

Apply component inverse_transform methods to estimator predictions in reverse order.

Components that implement inverse_transform are PolynomialDecomposer, LogTransformer, LabelEncoder (tbd).

**Parameters** `y` (*pd.Series*) – Final component features.

**Returns** The inverse transform of the target.

**Return type** pd.Series

**static load**(*file_path*)

Loads pipeline at file path.

**Parameters** **file_path** (*str*) – Location to load file.

**Returns** PipelineBase object

**property model_family**(*self*)

Returns model family of this pipeline.

**property name**(*self*)

Name of the pipeline.

**new**(*self*, *parameters*, *random_seed=0*)

Constructs a new instance of the pipeline with the same component graph but with a different set of parameters. Not to be confused with python's __new__ method.

**Parameters**

- **parameters** (*dict*) – Dictionary with component names as keys and dictionary of that component's parameters as values. An empty dictionary or None implies using all default values for component parameters. Defaults to None.
- **random_seed** (*int*) – Seed for the random number generator. Defaults to 0.

**Returns** A new instance of this pipeline with identical components.

**property parameters**(*self*)

Parameter dictionary for this pipeline.

**Returns** Dictionary of all component parameters.

**Return type** dict

**predict**(*self*, *X*, *objective=None*, *X_train=None*, *y_train=None*)

Predict on future data where target is not known.

**Parameters**

- **X** (*pd.DataFrame, or np.ndarray*) – Data of shape [n_samples, n_features].
- **objective** (*Object or string*) – The objective to use to make predictions.
- **X_train** (*pd.DataFrame or np.ndarray or None*) – Training data.
- **y_train** (*pd.Series or None*) – Training labels.

**Raises** **ValueError** – If X_train and/or y_train are None or if final component is not an Estimator.

**Returns** Predictions.

**predict_in_sample**(*self*, *X*, *y*, *X_train*, *y_train*, *objective=None*)

Predict on future data where the target is known, e.g. cross validation.

Note: we cast y as ints first to address boolean values that may be returned from calculating predictions which we would not be able to otherwise transform if we originally had integer targets.

**Parameters**

- **X** (*pd.DataFrame or np.ndarray*) – Future data of shape [n_samples, n_features].

- **y** (*pd.Series, np.ndarray*) – Future target of shape [n_samples].
- **X_train** (*pd.DataFrame, np.ndarray*) – Data the pipeline was trained on of shape [n_samples_train, n_features].
- **y_train** (*pd.Series, np.ndarray*) – Targets used to train the pipeline of shape [n_samples_train].
- **objective** (*ObjectiveBase, str, None*) – Objective used to threshold predicted probabilities, optional.

**Returns** Estimated labels.

**Return type** pd.Series

**Raises** **ValueError** – If final component is not an Estimator.

**predict_proba**(*self*, *X*, *X_train=None*, *y_train=None*)

Predict on future data where the target is unknown.

**Parameters**

- **X** (*pd.DataFrame or np.ndarray*) – Future data of shape [n_samples, n_features].
- **X_train** (*pd.DataFrame, np.ndarray*) – Data the pipeline was trained on of shape [n_samples_train, n_features].
- **y_train** (*pd.Series, np.ndarray*) – Targets used to train the pipeline of shape [n_samples_train].

**Returns** Estimated probabilities.

**Return type** pd.Series

**Raises** **ValueError** – If final component is not an Estimator.

**predict_proba_in_sample**(*self*, *X_holdout*, *y_holdout*, *X_train*, *y_train*)

Predict on future data where the target is known, e.g. cross validation.

**Parameters**

- **X_holdout** (*pd.DataFrame or np.ndarray*) – Future data of shape [n_samples, n_features].
- **y_holdout** (*pd.Series, np.ndarray*) – Future target of shape [n_samples].
- **X_train** (*pd.DataFrame, np.ndarray*) – Data the pipeline was trained on of shape [n_samples_train, n_features].
- **y_train** (*pd.Series, np.ndarray*) – Targets used to train the pipeline of shape [n_samples_train].

**Returns** Estimated probabilities.

**Return type** pd.Series

**Raises** **ValueError** – If the final component is not an Estimator.

**save**(*self*, *file_path*, *pickle_protocol=cloudpickle.DEFAULT_PROTOCOL*)

Saves pipeline at file path.

**Parameters**

- **file_path** (*str*) – Location to save file.
- **pickle_protocol** (*int*) – The pickle data stream format.

**score**(*self*, *X*, *y*, *objectives*, *X_train=None*, *y_train=None*)

Evaluate model performance on current and additional objectives.

**Parameters**

- **X** (*pd.DataFrame or np.ndarray*) – Data of shape [n_samples, n_features].
- **y** (*pd.Series*) – True labels of length [n_samples].
- **objectives** (*list*) – Non-empty list of objectives to score on.
- **X_train** (*pd.DataFrame, np.ndarray*) – Data the pipeline was trained on of shape [n_samples_train, n_features].
- **y_train** (*pd.Series, np.ndarray*) – Targets used to train the pipeline of shape [n_samples_train].

**Returns** Ordered dictionary of objective scores.

**Return type** dict

**property summary**(*self*)

A short summary of the pipeline structure, describing the list of components used.

Example: Logistic Regression Classifier w/ Simple Imputer + One Hot Encoder

**Returns** A string describing the pipeline structure.

**transform**(*self*, *X*, *y=None*)

Transform the input.

**Parameters**

- **X** (*pd.DataFrame, or np.ndarray*) – Data of shape [n_samples, n_features].
- **y** (*pd.Series*) – The target data of length [n_samples]. Defaults to None.

**Returns** Transformed output.

**Return type** pd.DataFrame

**transform_all_but_final**(*self*, *X*, *y=None*, *X_train=None*, *y_train=None*, *calculating_residuals=False*)

Transforms the data by applying all pre-processing components.

**Parameters**

- **X** (*pd.DataFrame*) – Input data to the pipeline to transform.
- **y** (*pd.Series*) – Targets corresponding to the pipeline targets.
- **X_train** (*pd.DataFrame*) – Training data used to generate generates from past observations.
- **y_train** (*pd.Series*) – Training targets used to generate features from past observations.
- **calculating_residuals** (*bool*) – Whether we're calling predict_in_sample to calculate the residuals. This means the X and y arguments are not future data, but actually the train data.

**Returns** New transformed features.

**Return type** pd.DataFrame

**class** evalml.pipelines.**TimeSeriesRegressionPipeline**(*component_graph*, *parameters=None*, *custom_name=None*, *random_seed=0*)

Pipeline base class for time series regression problems.

**Parameters**

- **component_graph** (ComponentGraph, *list*, *dict*) – ComponentGraph instance, list of components in order, or dictionary of components. Accepts strings or ComponentBase subclasses in the list. Note that when duplicate components are specified in a list, the duplicate component names will be modified with the component's index in the list. For example, the component graph [Imputer, One Hot Encoder, Imputer, Logistic Regression Classifier] will have names ["Imputer", "One Hot Encoder", "Imputer_2", "Logistic Regression Classifier"]
- **parameters** (*dict*) – Dictionary with component names as keys and dictionary of that component's parameters as values. An empty dictionary {} implies using all default values for component parameters. Pipeline-level parameters such as time_index, gap, and max_delay must be specified with the "pipeline" key. For example: Pipeline(parameters={"pipeline": {"time_index": "Date", "max_delay": 4, "gap": 2}}).
- **random_seed** (*int*) – Seed for the random number generator. Defaults to 0.

### Example

```
>>> pipeline = TimeSeriesRegressionPipeline(component_graph=["Simple Imputer",
↪"Linear Regressor"],
...                                          parameters={"Simple␣
↪Imputer": {"impute_strategy": "mean"},
...                                                      "pipeline": {
↪"gap": 1, "max_delay": 1, "forecast_horizon": 1, "time_index": "date"}},
...                                          custom_name="My␣
↪TimeSeriesRegression Pipeline")
...
>>> assert pipeline.custom_name == "My TimeSeriesRegression Pipeline"
>>> assert pipeline.component_graph.component_dict.keys() == {'Simple Imputer',
↪'Linear Regressor'}
```

The pipeline parameters will be chosen from the default parameters for every component, unless specific parameters were passed in as they were above.

```
>>> assert pipeline.parameters == {
...     'Simple Imputer': {'impute_strategy': 'mean', 'fill_value': None},
...     'Linear Regressor': {'fit_intercept': True, 'n_jobs': -1},
...     'pipeline': {'gap': 1, 'max_delay': 1, 'forecast_horizon': 1, 'time_index':
↪"date"}}
```

**Attributes**

| | |
|---|---|
| **NO_PREDS_PIPELINE_TYPES** | [PROBLEM_TYPE.TIME_SERIES_REGRESSION] |
| **problem_type** | None |

**Methods**

| | |
|---|---|
| *can_tune_threshold_with_objective* | Determine whether the threshold of a binary classification pipeline can be tuned. |
| *clone* | Constructs a new pipeline with the same components, parameters, and random seed. |
| *create_objectives* | Create objective instances from a list of strings or objective classes. |
| *custom_name* | Custom name of the pipeline. |
| *dates_needed_for_prediction* | Return dates needed to forecast the given date in the future. |
| *dates_needed_for_prediction_range* | Return dates needed to forecast the given date in the future. |
| *describe* | Outputs pipeline details including component parameters. |
| *feature_importance* | Importance associated with each feature. Features dropped by the feature selection are excluded. |
| *fit* | Fit a time series pipeline. |
| *fit_transform* | Fit and transform all components in the component graph, if all components are Transformers. |
| *get_component* | Returns component by name. |
| *get_forecast_period* | Generates all possible forecasting time points based on latest data point in X. |
| *get_forecast_predictions* | Generates all possible forecasting predictions based on last period of X. |
| *get_hyperparameter_ranges* | Returns hyperparameter ranges from all components as a dictionary. |
| *get_prediction_intervals* | Find the prediction intervals using the fitted regressor. |
| *graph* | Generate an image representing the pipeline graph. |
| *graph_dict* | Generates a dictionary with nodes consisting of the component names and parameters, and edges detailing component relationships. This dictionary is JSON serializable in most cases. |
| *graph_feature_importance* | Generate a bar graph of the pipeline's feature importance. |
| *inverse_transform* | Apply component inverse_transform methods to estimator predictions in reverse order. |
| *load* | Loads pipeline at file path. |
| *model_family* | Returns model family of this pipeline. |
| *name* | Name of the pipeline. |
| *new* | Constructs a new instance of the pipeline with the same component graph but with a different set of parameters. Not to be confused with python's __new__ method. |
| *parameters* | Parameter dictionary for this pipeline. |
| *predict* | Predict on future data where target is not known. |
| *predict_in_sample* | Predict on future data where the target is known, e.g. cross validation. |
| *save* | Saves pipeline at file path. |
| *score* | Evaluate model performance on current and additional objectives. |

continues on next page

Table 15 – continued from previous page

| | |
|---|---|
| *summary* | A short summary of the pipeline structure, describing the list of components used. |
| *transform* | Transform the input. |
| *transform_all_but_final* | Transforms the data by applying all pre-processing components. |

**can_tune_threshold_with_objective**(*self*, *objective*)

Determine whether the threshold of a binary classification pipeline can be tuned.

**Parameters** **objective** (*ObjectiveBase*) – Primary AutoMLSearch objective.

**Returns** True if the pipeline threshold can be tuned.

**Return type** bool

**clone**(*self*)

Constructs a new pipeline with the same components, parameters, and random seed.

**Returns** A new instance of this pipeline with identical components, parameters, and random seed.

**static create_objectives**(*objectives*)

Create objective instances from a list of strings or objective classes.

**property custom_name**(*self*)

Custom name of the pipeline.

**dates_needed_for_prediction**(*self*, *date*)

Return dates needed to forecast the given date in the future.

**Parameters** **date** (*pd.Timestamp*) – Date to forecast in the future.

**Returns** Range of dates needed to forecast the given date.

**Return type** dates_needed (tuple(pd.Timestamp))

**dates_needed_for_prediction_range**(*self*, *start_date*, *end_date*)

Return dates needed to forecast the given date in the future.

**Parameters**

- **start_date** (*pd.Timestamp*) – Start date of range to forecast in the future.
- **end_date** (*pd.Timestamp*) – End date of range to forecast in the future.

**Returns** Range of dates needed to forecast the given date.

**Return type** dates_needed (tuple(pd.Timestamp))

**Raises** **ValueError** – If start_date doesn't come before end_date

**describe**(*self*, *return_dict=False*)

Outputs pipeline details including component parameters.

**Parameters** **return_dict** (*bool*) – If True, return dictionary of information about pipeline. Defaults to False.

**Returns** Dictionary of all component parameters if return_dict is True, else None.

**Return type** dict

**property feature_importance**(*self*)

Importance associated with each feature. Features dropped by the feature selection are excluded.

**Returns** Feature names and their corresponding importance

**Return type** pd.DataFrame

**fit**(*self*, *X*, *y*)

Fit a time series pipeline.

**Parameters**

- **X** (*pd.DataFrame or np.ndarray*) – The input training data of shape [n_samples, n_features].
- **y** (*pd.Series, np.ndarray*) – The target training targets of length [n_samples].

**Returns** self

**Raises** **ValueError** – If the target is not numeric.

**fit_transform**(*self*, *X*, *y*)

Fit and transform all components in the component graph, if all components are Transformers.

**Parameters**

- **X** (*pd.DataFrame*) – Input features of shape [n_samples, n_features].
- **y** (*pd.Series*) – The target data of length [n_samples].

**Returns** Transformed output.

**Return type** pd.DataFrame

**Raises** **ValueError** – If final component is an Estimator.

**get_component**(*self*, *name*)

Returns component by name.

**Parameters** **name** (*str*) – Name of component.

**Returns** Component to return

**Return type** Component

**get_forecast_period**(*self*, *X*)

Generates all possible forecasting time points based on latest data point in X.

**Parameters** **X** (*pd.DataFrame, np.ndarray*) – Data the pipeline was trained on of shape [n_samples_train, n_feautures].

**Raises** **ValueError** – If pipeline is not trained.

**Returns** Datetime periods out to *forecast_horizon* + *gap*.

**Return type** pd.Series

### Example

```
>>> X = pd.DataFrame({'date': pd.date_range(start='1-1-2022', periods=10, freq=
↪'D'), 'feature': range(10, 20)})
>>> y = pd.Series(range(0, 10), name='target')
>>> gap = 1
>>> forecast_horizon = 2
>>> pipeline = TimeSeriesRegressionPipeline(component_graph=["Linear Regressor
↪"],
...                                          parameters={"Simple Imputer": {
↪"impute_strategy": "mean"},
...                                                      "pipeline": {"gap": gap,
↪ "max_delay": 1, "forecast_horizon": forecast_horizon, "time_index": "date"}},
...                                         )
>>> pipeline.fit(X, y)
pipeline = TimeSeriesRegressionPipeline(component_graph={'Linear Regressor': [
↪'Linear Regressor', 'X', 'y']}, parameters={'Linear Regressor':{'fit_intercept
↪': True, 'n_jobs': -1}, 'pipeline':{'gap': 1, 'max_delay': 1, 'forecast_
↪horizon': 2, 'time_index': 'date'}}, random_seed=0)
>>> dates = pipeline.get_forecast_period(X)
>>> expected = pd.Series(pd.date_range(start='2022-01-11', periods=(gap +␣
↪forecast_horizon), freq='D'), name='date', index=[10, 11, 12])
>>> assert dates.equals(expected)
```

**get_forecast_predictions**(*self*, *X*, *y*)

Generates all possible forecasting predictions based on last period of X.

**Parameters**

- **X** (*pd.DataFrame, np.ndarray*) – Data the pipeline was trained on of shape [n_samples_train, n_feautures].
- **y** (*pd.Series, np.ndarray*) – Targets used to train the pipeline of shape [n_samples_train].

**Returns** Predictions out to *forecast_horizon* + *gap* periods.

**get_hyperparameter_ranges**(*self*, *custom_hyperparameters*)

Returns hyperparameter ranges from all components as a dictionary.

**Parameters** **custom_hyperparameters** (*dict*) – Custom hyperparameters for the pipeline.

**Returns** Dictionary of hyperparameter ranges for each component in the pipeline.

**Return type** dict

**get_prediction_intervals**(*self*, *X*, *y=None*, *X_train=None*, *y_train=None*, *coverage=None*)

Find the prediction intervals using the fitted regressor.

This function takes the predictions of the fitted estimator and calculates the rolling standard deviation across all predictions using a window size of 5. The lower and upper predictions are determined by taking the percent point (quantile) function of the lower tail probability at each bound multiplied by the rolling standard deviation.

Certain estimators (Extra Trees Estimator, XGBoost Estimator, Prophet Estimator, ARIMA, and Exponential Smoothing estimator) utilize a different methodology to calculate prediction intervals. See the docs for these estimators to learn more.

**Parameters**

- **X** (*pd.DataFrame*) – Data of shape [n_samples, n_features].
- **y** (*pd.Series*) – Target data.
- **X_train** (*pd.DataFrame, np.ndarray*) – Data the pipeline was trained on of shape [n_samples_train, n_features].
- **y_train** (*pd.Series, np.ndarray*) – Targets used to train the pipeline of shape [n_samples_train].
- **coverage** (*list[float]*) – A list of floats between the values 0 and 1 that the upper and lower bounds of the prediction interval should be calculated for.

**Returns** Prediction intervals, keys are in the format {coverage}_lower or {coverage}_upper.

**Return type** dict

**Raises** **MethodPropertyNotFoundError** – If the estimator does not support Time Series Regression as a problem type.

**graph**(*self*, *filepath=None*)

Generate an image representing the pipeline graph.

**Parameters** **filepath** (*str, optional*) – Path to where the graph should be saved. If set to None (as by default), the graph will not be saved.

**Returns** Graph object that can be directly displayed in Jupyter notebooks.

**Return type** graphviz.Digraph

**Raises**

- **RuntimeError** – If graphviz is not installed.
- **ValueError** – If path is not writeable.

**graph_dict**(*self*)

Generates a dictionary with nodes consisting of the component names and parameters, and edges detailing component relationships. This dictionary is JSON serializable in most cases.

x_edges specifies from which component feature data is being passed. y_edges specifies from which component target data is being passed. This can be used to build graphs across a variety of visualization tools. Template: {"Nodes": {"component_name": {"Name": class_name, "Parameters": parameters_attributes}, ...}}, "x_edges": [[from_component_name, to_component_name], [from_component_name, to_component_name], ...], "y_edges": [[from_component_name, to_component_name], [from_component_name, to_component_name], ...]}

**Returns** A dictionary representing the DAG structure.

**Return type** dag_dict (dict)

**graph_feature_importance**(*self*, *importance_threshold=0*)

Generate a bar graph of the pipeline's feature importance.

**Parameters** **importance_threshold** (*float, optional*) – If provided, graph features with a permutation importance whose absolute value is larger than importance_threshold. Defaults to zero.

**Returns** A bar graph showing features and their corresponding importance.

**Return type** plotly.Figure

**Raises** **ValueError** – If importance threshold is not valid.

**inverse_transform**(*self*, *y*)

Apply component inverse_transform methods to estimator predictions in reverse order.

Components that implement inverse_transform are PolynomialDecomposer, LogTransformer, LabelEncoder (tbd).

**Parameters** **y** (*pd.Series*) – Final component features.

**Returns** The inverse transform of the target.

**Return type** pd.Series

**static load**(*file_path*)

Loads pipeline at file path.

**Parameters** **file_path** (*str*) – Location to load file.

**Returns** PipelineBase object

**property model_family**(*self*)

Returns model family of this pipeline.

**property name**(*self*)

Name of the pipeline.

**new**(*self*, *parameters*, *random_seed=0*)

Constructs a new instance of the pipeline with the same component graph but with a different set of parameters. Not to be confused with python's __new__ method.

**Parameters**

- **parameters** (*dict*) – Dictionary with component names as keys and dictionary of that component's parameters as values. An empty dictionary or None implies using all default values for component parameters. Defaults to None.
- **random_seed** (*int*) – Seed for the random number generator. Defaults to 0.

**Returns** A new instance of this pipeline with identical components.

**property parameters**(*self*)

Parameter dictionary for this pipeline.

**Returns** Dictionary of all component parameters.

**Return type** dict

**predict**(*self*, *X*, *objective=None*, *X_train=None*, *y_train=None*)

Predict on future data where target is not known.

**Parameters**

- **X** (*pd.DataFrame, or np.ndarray*) – Data of shape [n_samples, n_features].
- **objective** (*Object or string*) – The objective to use to make predictions.
- **X_train** (*pd.DataFrame or np.ndarray or None*) – Training data.
- **y_train** (*pd.Series or None*) – Training labels.

**Raises** **ValueError** – If X_train and/or y_train are None or if final component is not an Estimator.

**Returns** Predictions.

**predict_in_sample**(*self*, *X*, *y*, *X_train*, *y_train*, *objective=None*, *calculating_residuals=False*)

Predict on future data where the target is known, e.g. cross validation.

**Parameters**

- **X** (*pd.DataFrame or np.ndarray*) – Future data of shape [n_samples, n_features]
- **y** (*pd.Series, np.ndarray*) – Future target of shape [n_samples]
- **X_train** (*pd.DataFrame, np.ndarray*) – Data the pipeline was trained on of shape [n_samples_train, n_feautures]
- **y_train** (*pd.Series, np.ndarray*) – Targets used to train the pipeline of shape [n_samples_train]
- **objective** (*ObjectiveBase, str, None*) – Objective used to threshold predicted probabilities, optional.
- **calculating_residuals** (*bool*) – Whether we're calling predict_in_sample to calculate the residuals. This means the X and y arguments are not future data, but actually the train data.

**Returns** Estimated labels.

**Return type** pd.Series

**Raises** **ValueError** – If final component is not an Estimator.

**save**(*self*, *file_path*, *pickle_protocol=cloudpickle.DEFAULT_PROTOCOL*)

Saves pipeline at file path.

**Parameters**

- **file_path** (*str*) – Location to save file.
- **pickle_protocol** (*int*) – The pickle data stream format.

**score**(*self*, *X*, *y*, *objectives*, *X_train=None*, *y_train=None*)

Evaluate model performance on current and additional objectives.

**Parameters**

- **X** (*pd.DataFrame or np.ndarray*) – Data of shape [n_samples, n_features].
- **y** (*pd.Series*) – True labels of length [n_samples].
- **objectives** (*list*) – Non-empty list of objectives to score on.
- **X_train** (*pd.DataFrame, np.ndarray*) – Data the pipeline was trained on of shape [n_samples_train, n_feautures].
- **y_train** (*pd.Series, np.ndarray*) – Targets used to train the pipeline of shape [n_samples_train].

**Returns** Ordered dictionary of objective scores.

**Return type** dict

**property summary**(*self*)

A short summary of the pipeline structure, describing the list of components used.

Example: Logistic Regression Classifier w/ Simple Imputer + One Hot Encoder

**Returns** A string describing the pipeline structure.

**transform**(*self*, *X*, *y=None*)

Transform the input.

**Parameters**

- **X** (*pd.DataFrame, or np.ndarray*) – Data of shape [n_samples, n_features].
- **y** (*pd.Series*) – The target data of length [n_samples]. Defaults to None.

**Returns** Transformed output.

**Return type** pd.DataFrame

**transform_all_but_final**(*self*, *X*, *y=None*, *X_train=None*, *y_train=None*, *calculating_residuals=False*)

Transforms the data by applying all pre-processing components.

**Parameters**

- **X** (*pd.DataFrame*) – Input data to the pipeline to transform.
- **y** (*pd.Series*) – Targets corresponding to the pipeline targets.
- **X_train** (*pd.DataFrame*) – Training data used to generate generates from past observations.
- **y_train** (*pd.Series*) – Training targets used to generate features from past observations.
- **calculating_residuals** (*bool*) – Whether we're calling predict_in_sample to calculate the residuals. This means the X and y arguments are not future data, but actually the train data.

**Returns** New transformed features.

**Return type** pd.DataFrame

**class** evalml.pipelines.**TimeSeriesRegularizer**(*time_index=None*, *frequency_payload=None*, *window_length=4*, *threshold=0.4*, *random_seed=0*, ***kwargs*)

Transformer that regularizes an inconsistently spaced datetime column.

If X is passed in to fit/transform, the column *time_index* will be checked for an inferrable offset frequency. If the *time_index* column is perfectly inferrable then this Transformer will do nothing and return the original X and y.

If X does not have a perfectly inferrable frequency but one can be estimated, then X and y will be reformatted based on the estimated frequency for *time_index*. In the original X and y passed: - Missing datetime values will be added and will have their corresponding columns in X and y set to None. - Duplicate datetime values will be dropped. - Extra datetime values will be dropped. - If it can be determined that a duplicate or extra value is misaligned, then it will be repositioned to take the place of a missing value.

This Transformer should be used before the *TimeSeriesImputer* in order to impute the missing values that were added to X and y (if passed).

**Parameters**

- **time_index** (*string*) – Name of the column containing the datetime information used to order the data, required. Defaults to None.
- **frequency_payload** (*tuple*) – Payload returned from Woodwork's infer_frequency function where debug is True. Defaults to None.
- **window_length** (*int*) – The size of the rolling window over which inference is conducted to determine the prevalence of uninferrable frequencies.
- **5.** (*Lower values make this component more sensitive to recognizing numerous faulty datetime values. Defaults to*) –

- **threshold** (*float*) – The minimum percentage of windows that need to have been able to infer a frequency. Lower values make this component more
- **0.8.** (*sensitive to recognizing numerous faulty datetime values. Defaults to*) –
- **random_seed** (*int*) – Seed for the random number generator. This transformer performs the same regardless of the random seed provided.
- **0.** (*Defaults to*) –

**Raises** **ValueError** – if the frequency_payload parameter has not been passed a tuple

**Attributes**

| | |
|---|---|
| **hyper-parameter_ranges** | {} |
| **modifies_features** | True |
| **modifies_target** | True |
| **name** | Time Series Regularizer |
| **training_only** | True |

**Methods**

| | |
|---|---|
| *clone* | Constructs a new component with the same parameters and random state. |
| *default_parameters* | Returns the default parameters for this component. |
| *describe* | Describe a component and its parameters. |
| *fit* | Fits the TimeSeriesRegularizer. |
| *fit_transform* | Fits on X and transforms X. |
| *load* | Loads component at file path. |
| *needs_fitting* | Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances. |
| *parameters* | Returns the parameters which were used to initialize the component. |
| *save* | Saves component at file path. |
| *transform* | Regularizes a dataframe and target data to an inferrable offset frequency. |
| *update_parameters* | Updates the parameter dictionary of the component. |

**clone**(*self*)

Constructs a new component with the same parameters and random state.

**Returns** A new instance of this component with identical parameters and random state.

**default_parameters**(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

**Returns** Default parameters for this component.

**Return type** dict

**describe**(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

**Parameters**

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description as dictionary in the format {"name": name, "parameters": parameters}

**Returns** Returns dictionary if return_dict is True, else None.

**Return type** None or dict

**fit**(*self*, *X*, *y=None*)

Fits the TimeSeriesRegularizer.

**Parameters**

- **X** (*pd.DataFrame*) – The input training data of shape [n_samples, n_features].
- **y** (*pd.Series, optional*) – The target training data of length [n_samples].

**Returns** self

**Raises**

- **ValueError** – if self.time_index is None, if X and y have different lengths, if *time_index* in X does not have an offset frequency that can be estimated
- **TypeError** – if the *time_index* column is not of type Datetime
- **KeyError** – if the *time_index* column doesn't exist

**fit_transform**(*self*, *X*, *y=None*)

Fits on X and transforms X.

**Parameters**

- **X** (*pd.DataFrame*) – Data to fit and transform.
- **y** (*pd.Series*) – Target data.

**Returns** Transformed X.

**Return type** pd.DataFrame

**Raises** **MethodPropertyNotFoundError** – If transformer does not have a transform method or a component_obj that implements transform.

**static load**(*file_path*)

Loads component at file path.

**Parameters** **file_path** (*str*) – Location to load file.

**Returns** ComponentBase object

**needs_fitting**(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need to be fit or whose fit methods do nothing.

**Returns** True.

**property parameters**(*self*)

Returns the parameters which were used to initialize the component.

**save**(*self*, *file_path*, *pickle_protocol=cloudpickle.DEFAULT_PROTOCOL*)

Saves component at file path.

**Parameters**

- **file_path** (*str*) – Location to save file.
- **pickle_protocol** (*int*) – The pickle data stream format.

**transform**(*self*, *X*, *y=None*)

Regularizes a dataframe and target data to an inferrable offset frequency.

A 'clean' X and y (if y was passed in) are created based on an inferrable offset frequency and matching datetime values with the original X and y are imputed into the clean X and y. Datetime values identified as misaligned are shifted into their appropriate position.

**Parameters**

- **X** (*pd.DataFrame*) – The input training data of shape [n_samples, n_features].
- **y** (*pd.Series, optional*) – The target training data of length [n_samples].

**Returns** Data with an inferrable *time_index* offset frequency.

**Return type** (pd.DataFrame, pd.Series)

**update_parameters**(*self*, *update_dict*, *reset_fit=True*)

Updates the parameter dictionary of the component.

**Parameters**

- **update_dict** (*dict*) – A dict of parameters to update.
- **reset_fit** (*bool, optional*) – If True, will set *_is_fitted* to False.

**class** evalml.pipelines.**Transformer**(*parameters=None*, *component_obj=None*, *random_seed=0*, ***kwargs*)

A component that may or may not need fitting that transforms data. These components are used before an estimator.

To implement a new Transformer, define your own class which is a subclass of Transformer, including a name and a list of acceptable ranges for any parameters to be tuned during the automl search (hyperparameters). Define an *__init__* method which sets up any necessary state and objects. Make sure your *__init__* only uses standard keyword arguments and calls *super().__init__()* with a parameters dict. You may also override the *fit*, *transform*, *fit_transform* and other methods in this class if appropriate.

To see some examples, check out the definitions of any Transformer component.

**Parameters**

- **parameters** (*dict*) – Dictionary of parameters for the component. Defaults to None.
- **component_obj** (*obj*) – Third-party objects useful in component implementation. Defaults to None.
- **random_seed** (*int*) – Seed for the random number generator. Defaults to 0.

**Attributes**

| | |
|---|---|
| **modifies_features** | True |
| **modifies_target** | False |
| **training_only** | False |

**Methods**

| | |
|---|---|
| *clone* | Constructs a new component with the same parameters and random state. |
| *default_parameters* | Returns the default parameters for this component. |
| *describe* | Describe a component and its parameters. |
| *fit* | Fits component to data. |
| *fit_transform* | Fits on X and transforms X. |
| *load* | Loads component at file path. |
| *name* | Returns string name of this component. |
| *needs_fitting* | Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances. |
| *parameters* | Returns the parameters which were used to initialize the component. |
| *save* | Saves component at file path. |
| *transform* | Transforms data X. |
| *update_parameters* | Updates the parameter dictionary of the component. |

**clone**(*self*)

Constructs a new component with the same parameters and random state.

**Returns** A new instance of this component with identical parameters and random state.

**default_parameters**(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

**Returns** Default parameters for this component.

**Return type** dict

**describe**(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

**Parameters**

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description as dictionary in the format {"name": name, "parameters": parameters}

**Returns** Returns dictionary if return_dict is True, else None.

**Return type** None or dict

**fit**(*self*, *X*, *y=None*)

Fits component to data.

**Parameters**

- **X** (*pd.DataFrame*) – The input training data of shape [n_samples, n_features]
- **y** (*pd.Series, optional*) – The target training data of length [n_samples]

**Returns** self

**Raises** **MethodPropertyNotFoundError** – If component does not have a fit method or a component_obj that implements fit.

**fit_transform**(*self*, *X*, *y=None*)

Fits on X and transforms X.

**Parameters**

- **X** (*pd.DataFrame*) – Data to fit and transform.
- **y** (*pd.Series*) – Target data.

**Returns** Transformed X.

**Return type** pd.DataFrame

**Raises** **MethodPropertyNotFoundError** – If transformer does not have a transform method or a component_obj that implements transform.

**static load**(*file_path*)

Loads component at file path.

**Parameters** **file_path** (*str*) – Location to load file.

**Returns** ComponentBase object

**property name**(*cls*)

Returns string name of this component.

**needs_fitting**(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need to be fit or whose fit methods do nothing.

**Returns** True.

**property parameters**(*self*)

Returns the parameters which were used to initialize the component.

**save**(*self*, *file_path*, *pickle_protocol=cloudpickle.DEFAULT_PROTOCOL*)

Saves component at file path.

**Parameters**

- **file_path** (*str*) – Location to save file.
- **pickle_protocol** (*int*) – The pickle data stream format.

**abstract transform**(*self*, *X*, *y=None*)

Transforms data X.

**Parameters**

- **X** (*pd.DataFrame*) – Data to transform.
- **y** (*pd.Series, optional*) – Target data.

**Returns** Transformed X

**Return type** pd.DataFrame

**Raises** `MethodPropertyNotFoundError` – If transformer does not have a transform method or a component_obj that implements transform.

`update_parameters`(*self*, *update_dict*, *reset_fit=True*)

Updates the parameter dictionary of the component.

**Parameters**

- `update_dict` (*dict*) – A dict of parameters to update.
- `reset_fit` (*bool, optional*) – If True, will set *_is_fitted* to False.

class evalml.pipelines.`VowpalWabbitBinaryClassifier`(*loss_function='logistic'*, *learning_rate=0.5*, *decay_learning_rate=1.0*, *power_t=0.5*, *passes=1*, *random_seed=0*, ***kwargs*)

Vowpal Wabbit Binary Classifier.

**Parameters**

- `loss_function` (*str*) – Specifies the loss function to use. One of {"squared", "classic", "hinge", "logistic", "quantile"}. Defaults to "logistic".
- `learning_rate` (*float*) – Boosting learning rate. Defaults to 0.5.
- `decay_learning_rate` (*float*) – Decay factor for learning_rate. Defaults to 1.0.
- `power_t` (*float*) – Power on learning rate decay. Defaults to 0.5.
- `passes` (*int*) – Number of training passes. Defaults to 1.
- `random_seed` (*int*) – Seed for the random number generator. Defaults to 0.

**Attributes**

| | |
|---|---|
| **hyper-parameter_ranges** | None |
| **model_family** | ModelFamily.VOWPAL_WABBIT |
| **modifies_features** | True |
| **modifies_target** | False |
| **name** | Vowpal Wabbit Binary Classifier |
| **supported_problem_types** | [ ProblemTypes.BINARY, ProblemTypes.TIME_SERIES_BINARY,] |
| **training_only** | False |

**Methods**

| | |
|---|---|
| *clone* | Constructs a new component with the same parameters and random state. |
| *default_parameters* | Returns the default parameters for this component. |
| *describe* | Describe a component and its parameters. |
| *feature_importance* | Feature importance for Vowpal Wabbit classifiers. This is not implemented. |
| *fit* | Fits estimator to data. |
| *get_prediction_intervals* | Find the prediction intervals using the fitted regressor. |
| *load* | Loads component at file path. |
| *needs_fitting* | Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances. |
| *parameters* | Returns the parameters which were used to initialize the component. |
| *predict* | Make predictions using selected features. |
| *predict_proba* | Make probability estimates for labels. |
| *save* | Saves component at file path. |
| *update_parameters* | Updates the parameter dictionary of the component. |

**clone**(*self*)

Constructs a new component with the same parameters and random state.

**Returns** A new instance of this component with identical parameters and random state.

**default_parameters**(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

**Returns** Default parameters for this component.

**Return type** dict

**describe**(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

**Parameters**

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description as dictionary in the format {"name": name, "parameters": parameters}

**Returns** Returns dictionary if return_dict is True, else None.

**Return type** None or dict

**property feature_importance**(*self*)

Feature importance for Vowpal Wabbit classifiers. This is not implemented.

**fit**(*self*, *X: pandas.DataFrame*, *y: Optional[pandas.Series] = None*)

Fits estimator to data.

**Parameters**

- **X** (*pd.DataFrame*) – The input training data of shape [n_samples, n_features].
- **y** (*pd.Series, optional*) – The target training data of length [n_samples].

**Returns** self

**get_prediction_intervals**(*self*, *X: pandas.DataFrame*, *y: Optional[pandas.Series] = None*, *coverage: List[float] = None*, *predictions: pandas.Series = None*) → Dict[str, pandas.Series]

Find the prediction intervals using the fitted regressor.

This function takes the predictions of the fitted estimator and calculates the rolling standard deviation across all predictions using a window size of 5. The lower and upper predictions are determined by taking the percent point (quantile) function of the lower tail probability at each bound multiplied by the rolling standard deviation.

**Parameters**

- **X** (*pd.DataFrame*) – Data of shape [n_samples, n_features].
- **y** (*pd.Series*) – Target data. Ignored.
- **coverage** (*list[float]*) – A list of floats between the values 0 and 1 that the upper and lower bounds of the prediction interval should be calculated for.
- **predictions** (*pd.Series*) – Optional list of predictions to use. If None, will generate predictions using *X*.

**Returns** Prediction intervals, keys are in the format {coverage}_lower or {coverage}_upper.

**Return type** dict

**Raises** **MethodPropertyNotFoundError** – If the estimator does not support Time Series Regression as a problem type.

**static load**(*file_path*)

Loads component at file path.

**Parameters** **file_path** (*str*) – Location to load file.

**Returns** ComponentBase object

**needs_fitting**(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need to be fit or whose fit methods do nothing.

**Returns** True.

**property parameters**(*self*)

Returns the parameters which were used to initialize the component.

**predict**(*self*, *X: pandas.DataFrame*) → pandas.Series

Make predictions using selected features.

**Parameters** **X** (*pd.DataFrame*) – Data of shape [n_samples, n_features].

**Returns** Predicted values.

**Return type** pd.Series

**Raises** **MethodPropertyNotFoundError** – If estimator does not have a predict method or a component_obj that implements predict.

**predict_proba**(*self*, *X: pandas.DataFrame*) → pandas.Series

Make probability estimates for labels.

**Parameters** **X** (*pd.DataFrame*) – Features.

**Returns** Probability estimates.

**Return type** pd.Series

**Raises** **MethodPropertyNotFoundError** – If estimator does not have a predict_proba method or a component_obj that implements predict_proba.

**save**(*self*, *file_path*, *pickle_protocol=cloudpickle.DEFAULT_PROTOCOL*)

Saves component at file path.

**Parameters**

- **file_path** (*str*) – Location to save file.
- **pickle_protocol** (*int*) – The pickle data stream format.

**update_parameters**(*self*, *update_dict*, *reset_fit=True*)

Updates the parameter dictionary of the component.

**Parameters**

- **update_dict** (*dict*) – A dict of parameters to update.
- **reset_fit** (*bool, optional*) – If True, will set *_is_fitted* to False.

**class** evalml.pipelines.**VowpalWabbitMulticlassClassifier**(*loss_function='logistic'*, *learning_rate=0.5*, *decay_learning_rate=1.0*, *power_t=0.5*, *passes=1*, *random_seed=0*, ***kwargs*)

Vowpal Wabbit Multiclass Classifier.

**Parameters**

- **loss_function** (*str*) – Specifies the loss function to use. One of {"squared", "classic", "hinge", "logistic", "quantile"}. Defaults to "logistic".
- **learning_rate** (*float*) – Boosting learning rate. Defaults to 0.5.
- **decay_learning_rate** (*float*) – Decay factor for learning_rate. Defaults to 1.0.
- **power_t** (*float*) – Power on learning rate decay. Defaults to 0.5.
- **random_seed** (*int*) – Seed for the random number generator. Defaults to 0.

**Attributes**

| | |
|---|---|
| **hyper-parameter_ranges** | None |
| **model_family** | ModelFamily.VOWPAL_WABBIT |
| **modifies_features** | True |
| **modifies_target** | False |
| **name** | Vowpal Wabbit Multiclass Classifier |
| **supported_problem_types** | [ ProblemTypes.MULTICLASS, ProblemTypes.TIME_SERIES_MULTICLASS,] |
| **training_only** | False |

**Methods**

| | |
|---|---|
| *clone* | Constructs a new component with the same parameters and random state. |
| *default_parameters* | Returns the default parameters for this component. |
| *describe* | Describe a component and its parameters. |
| *feature_importance* | Feature importance for Vowpal Wabbit classifiers. This is not implemented. |
| *fit* | Fits estimator to data. |
| *get_prediction_intervals* | Find the prediction intervals using the fitted regressor. |
| *load* | Loads component at file path. |
| *needs_fitting* | Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances. |
| *parameters* | Returns the parameters which were used to initialize the component. |
| *predict* | Make predictions using selected features. |
| *predict_proba* | Make probability estimates for labels. |
| *save* | Saves component at file path. |
| *update_parameters* | Updates the parameter dictionary of the component. |

**clone**(*self*)

Constructs a new component with the same parameters and random state.

**Returns** A new instance of this component with identical parameters and random state.

**default_parameters**(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

**Returns** Default parameters for this component.

**Return type** dict

**describe**(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

**Parameters**

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description as dictionary in the format {"name": name, "parameters": parameters}

**Returns** Returns dictionary if return_dict is True, else None.

**Return type** None or dict

**property feature_importance**(*self*)

Feature importance for Vowpal Wabbit classifiers. This is not implemented.

**fit**(*self*, *X: pandas.DataFrame*, *y: Optional[pandas.Series] = None*)

Fits estimator to data.

**Parameters**

- **X** (*pd.DataFrame*) – The input training data of shape [n_samples, n_features].

- **y** (*pd.Series, optional*) – The target training data of length [n_samples].

**Returns** self

**get_prediction_intervals**(*self, X: pandas.DataFrame, y: Optional[pandas.Series] = None, coverage: List[float] = None, predictions: pandas.Series = None*) → Dict[str, pandas.Series]

Find the prediction intervals using the fitted regressor.

This function takes the predictions of the fitted estimator and calculates the rolling standard deviation across all predictions using a window size of 5. The lower and upper predictions are determined by taking the percent point (quantile) function of the lower tail probability at each bound multiplied by the rolling standard deviation.

**Parameters**

- **X** (*pd.DataFrame*) – Data of shape [n_samples, n_features].
- **y** (*pd.Series*) – Target data. Ignored.
- **coverage** (*list[float]*) – A list of floats between the values 0 and 1 that the upper and lower bounds of the prediction interval should be calculated for.
- **predictions** (*pd.Series*) – Optional list of predictions to use. If None, will generate predictions using *X*.

**Returns** Prediction intervals, keys are in the format {coverage}_lower or {coverage}_upper.

**Return type** dict

**Raises** **MethodPropertyNotFoundError** – If the estimator does not support Time Series Regression as a problem type.

**static load**(*file_path*)

Loads component at file path.

**Parameters** **file_path** (*str*) – Location to load file.

**Returns** ComponentBase object

**needs_fitting**(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need to be fit or whose fit methods do nothing.

**Returns** True.

**property parameters**(*self*)

Returns the parameters which were used to initialize the component.

**predict**(*self, X: pandas.DataFrame*) → pandas.Series

Make predictions using selected features.

**Parameters** **X** (*pd.DataFrame*) – Data of shape [n_samples, n_features].

**Returns** Predicted values.

**Return type** pd.Series

**Raises** **MethodPropertyNotFoundError** – If estimator does not have a predict method or a component_obj that implements predict.

**predict_proba**(*self*, *X: pandas.DataFrame*) → pandas.Series

Make probability estimates for labels.

**Parameters** **X** (*pd.DataFrame*) – Features.

**Returns** Probability estimates.

**Return type** pd.Series

**Raises** **MethodPropertyNotFoundError** – If estimator does not have a predict_proba method or a component_obj that implements predict_proba.

**save**(*self*, *file_path*, *pickle_protocol=cloudpickle.DEFAULT_PROTOCOL*)

Saves component at file path.

**Parameters**

- **file_path** (*str*) – Location to save file.
- **pickle_protocol** (*int*) – The pickle data stream format.

**update_parameters**(*self*, *update_dict*, *reset_fit=True*)

Updates the parameter dictionary of the component.

**Parameters**

- **update_dict** (*dict*) – A dict of parameters to update.
- **reset_fit** (*bool, optional*) – If True, will set *_is_fitted* to False.

**class** evalml.pipelines.**VowpalWabbitRegressor**(*learning_rate=0.5*, *decay_learning_rate=1.0*, *power_t=0.5*, *passes=1*, *random_seed=0*, ***kwargs*)

Vowpal Wabbit Regressor.

**Parameters**

- **learning_rate** (*float*) – Boosting learning rate. Defaults to 0.5.
- **decay_learning_rate** (*float*) – Decay factor for learning_rate. Defaults to 1.0.
- **power_t** (*float*) – Power on learning rate decay. Defaults to 0.5.
- **passes** (*int*) – Number of training passes. Defaults to 1.
- **random_seed** (*int*) – Seed for the random number generator. Defaults to 0.

**Attributes**

| | |
|---|---|
| **hyper-parameter_ranges** | None |
| **model_family** | ModelFamily.VOWPAL_WABBIT |
| **modifies_features** | True |
| **modifies_target** | False |
| **name** | Vowpal Wabbit Regressor |
| **supported_problem_types** | [ ProblemTypes.REGRESSION, ProblemTypes.TIME_SERIES_REGRESSION,] |
| **training_only** | False |

**Methods**

| | |
|---|---|
| *clone* | Constructs a new component with the same parameters and random state. |
| *default_parameters* | Returns the default parameters for this component. |
| *describe* | Describe a component and its parameters. |
| *feature_importance* | Feature importance for Vowpal Wabbit regressor. |
| *fit* | Fits estimator to data. |
| *get_prediction_intervals* | Find the prediction intervals using the fitted regressor. |
| *load* | Loads component at file path. |
| *needs_fitting* | Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances. |
| *parameters* | Returns the parameters which were used to initialize the component. |
| *predict* | Make predictions using selected features. |
| *predict_proba* | Make probability estimates for labels. |
| *save* | Saves component at file path. |
| *update_parameters* | Updates the parameter dictionary of the component. |

**clone**(*self*)

Constructs a new component with the same parameters and random state.

**Returns** A new instance of this component with identical parameters and random state.

**default_parameters**(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

**Returns** Default parameters for this component.

**Return type** dict

**describe**(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

**Parameters**

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description as dictionary in the format {"name": name, "parameters": parameters}

**Returns** Returns dictionary if return_dict is True, else None.

**Return type** None or dict

**property feature_importance**(*self*)

Feature importance for Vowpal Wabbit regressor.

**fit**(*self*, *X: pandas.DataFrame*, *y: Optional[pandas.Series] = None*)

Fits estimator to data.

**Parameters**

- **X** (*pd.DataFrame*) – The input training data of shape [n_samples, n_features].
- **y** (*pd.Series, optional*) – The target training data of length [n_samples].

**Returns** self

**get_prediction_intervals**(*self*, *X: pandas.DataFrame*, *y: Optional[pandas.Series] = None*, *coverage: List[float] = None*, *predictions: pandas.Series = None*) → Dict[str, pandas.Series]

Find the prediction intervals using the fitted regressor.

This function takes the predictions of the fitted estimator and calculates the rolling standard deviation across all predictions using a window size of 5. The lower and upper predictions are determined by taking the percent point (quantile) function of the lower tail probability at each bound multiplied by the rolling standard deviation.

**Parameters**

- **X** (*pd.DataFrame*) – Data of shape [n_samples, n_features].
- **y** (*pd.Series*) – Target data. Ignored.
- **coverage** (*list[float]*) – A list of floats between the values 0 and 1 that the upper and lower bounds of the prediction interval should be calculated for.
- **predictions** (*pd.Series*) – Optional list of predictions to use. If None, will generate predictions using *X*.

**Returns** Prediction intervals, keys are in the format {coverage}_lower or {coverage}_upper.

**Return type** dict

**Raises** **MethodPropertyNotFoundError** – If the estimator does not support Time Series Regression as a problem type.

**static load**(*file_path*)

Loads component at file path.

**Parameters** **file_path** (*str*) – Location to load file.

**Returns** ComponentBase object

**needs_fitting**(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need to be fit or whose fit methods do nothing.

**Returns** True.

**property parameters**(*self*)

Returns the parameters which were used to initialize the component.

**predict**(*self*, *X: pandas.DataFrame*) → pandas.Series

Make predictions using selected features.

**Parameters** **X** (*pd.DataFrame*) – Data of shape [n_samples, n_features].

**Returns** Predicted values.

**Return type** pd.Series

**Raises** **MethodPropertyNotFoundError** – If estimator does not have a predict method or a component_obj that implements predict.

**predict_proba**(*self*, *X: pandas.DataFrame*) → pandas.Series

Make probability estimates for labels.

**Parameters** **X** (*pd.DataFrame*) – Features.

**Returns** Probability estimates.

**Return type** pd.Series

**Raises** **MethodPropertyNotFoundError** – If estimator does not have a predict_proba method or a component_obj that implements predict_proba.

**save**(*self*, *file_path*, *pickle_protocol=cloudpickle.DEFAULT_PROTOCOL*)

Saves component at file path.

**Parameters**

- **file_path** (*str*) – Location to save file.
- **pickle_protocol** (*int*) – The pickle data stream format.

**update_parameters**(*self*, *update_dict*, *reset_fit=True*)

Updates the parameter dictionary of the component.

**Parameters**

- **update_dict** (*dict*) – A dict of parameters to update.
- **reset_fit** (*bool, optional*) – If True, will set *_is_fitted* to False.

class evalml.pipelines.**XGBoostClassifier**(*eta=0.1*, *max_depth=6*, *min_child_weight=1*, *n_estimators=100*, *random_seed=0*, *eval_metric='logloss'*, *n_jobs=12*, ***kwargs*)

XGBoost Classifier.

**Parameters**

- **eta** (*float*) – Boosting learning rate. Defaults to 0.1.
- **max_depth** (*int*) – Maximum tree depth for base learners. Defaults to 6.
- **min_child_weight** (*float*) – Minimum sum of instance weight (hessian) needed in a child. Defaults to 1.0
- **n_estimators** (*int*) – Number of gradient boosted trees. Equivalent to number of boosting rounds. Defaults to 100.
- **random_seed** (*int*) – Seed for the random number generator. Defaults to 0.
- **n_jobs** (*int*) – Number of parallel threads used to run xgboost. Note that creating thread contention will significantly slow down the algorithm. Defaults to 12.

**Attributes**

| | |
|---|---|
| **hyper-parameter_ranges** | { "eta": Real(0.000001, 1), "max_depth": Integer(1, 10), "min_child_weight": Real(1, 10), "n_estimators": Integer(1, 1000),} |
| **model_family** | ModelFamily.XGBOOST |
| **modifies_features** | True |
| **modifies_target** | False |
| **name** | XGBoost Classifier |
| **SEED_MAX** | None |
| **SEED_MIN** | None |
| **supported_problem_types** | [ ProblemTypes.BINARY, ProblemTypes.MULTICLASS, ProblemTypes.TIME_SERIES_BINARY, ProblemTypes.TIME_SERIES_MULTICLASS,] |
| **training_only** | False |

**Methods**

| | |
|---|---|
| *clone* | Constructs a new component with the same parameters and random state. |
| *default_parameters* | Returns the default parameters for this component. |
| *describe* | Describe a component and its parameters. |
| *feature_importance* | Feature importance of fitted XGBoost classifier. |
| *fit* | Fits XGBoost classifier component to data. |
| *get_prediction_intervals* | Find the prediction intervals using the fitted regressor. |
| *load* | Loads component at file path. |
| *needs_fitting* | Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances. |
| *parameters* | Returns the parameters which were used to initialize the component. |
| *predict* | Make predictions using the fitted XGBoost classifier. |
| *predict_proba* | Make predictions using the fitted CatBoost classifier. |
| *save* | Saves component at file path. |
| *update_parameters* | Updates the parameter dictionary of the component. |

**clone**(*self*)

Constructs a new component with the same parameters and random state.

**Returns** A new instance of this component with identical parameters and random state.

**default_parameters**(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

**Returns** Default parameters for this component.

**Return type** dict

**describe**(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

**Parameters**

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description as dictionary in the format {"name": name, "parameters": parameters}

**Returns** Returns dictionary if return_dict is True, else None.

**Return type** None or dict

**property feature_importance**(*self*)

Feature importance of fitted XGBoost classifier.

**fit**(*self*, *X*, *y=None*)

Fits XGBoost classifier component to data.

**Parameters**

- **X** (*pd.DataFrame*) – The input training data of shape [n_samples, n_features].
- **y** (*pd.Series*) – The target training data of length [n_samples].

**Returns** self

**get_prediction_intervals**(*self*, *X: pandas.DataFrame*, *y: Optional[pandas.Series] = None*, *coverage: List[float] = None*, *predictions: pandas.Series = None*) → Dict[str, pandas.Series]

Find the prediction intervals using the fitted regressor.

This function takes the predictions of the fitted estimator and calculates the rolling standard deviation across all predictions using a window size of 5. The lower and upper predictions are determined by taking the percent point (quantile) function of the lower tail probability at each bound multiplied by the rolling standard deviation.

**Parameters**

- **X** (*pd.DataFrame*) – Data of shape [n_samples, n_features].
- **y** (*pd.Series*) – Target data. Ignored.
- **coverage** (*list[float]*) – A list of floats between the values 0 and 1 that the upper and lower bounds of the prediction interval should be calculated for.
- **predictions** (*pd.Series*) – Optional list of predictions to use. If None, will generate predictions using *X*.

**Returns** Prediction intervals, keys are in the format {coverage}_lower or {coverage}_upper.

**Return type** dict

**Raises** **MethodPropertyNotFoundError** – If the estimator does not support Time Series Regression as a problem type.

**static load**(*file_path*)

Loads component at file path.

**Parameters** **file_path** (*str*) – Location to load file.

**Returns** ComponentBase object

**needs_fitting**(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need to be fit or whose fit methods do nothing.

**Returns** True.

**property parameters**(*self*)

Returns the parameters which were used to initialize the component.

**predict**(*self*, *X*)

Make predictions using the fitted XGBoost classifier.

**Parameters** **X** (*pd.DataFrame*) – Data of shape [n_samples, n_features].

**Returns** Predicted values.

**Return type** pd.DataFrame

**predict_proba**(*self*, *X*)

Make predictions using the fitted CatBoost classifier.

**Parameters** **X** (*pd.DataFrame*) – Data of shape [n_samples, n_features].

**Returns** Predicted values.

**Return type** pd.DataFrame

**save**(*self*, *file_path*, *pickle_protocol=cloudpickle.DEFAULT_PROTOCOL*)

Saves component at file path.

**Parameters**

- **file_path** (*str*) – Location to save file.
- **pickle_protocol** (*int*) – The pickle data stream format.

**update_parameters**(*self*, *update_dict*, *reset_fit=True*)

Updates the parameter dictionary of the component.

**Parameters**

- **update_dict** (*dict*) – A dict of parameters to update.
- **reset_fit** (*bool, optional*) – If True, will set *_is_fitted* to False.

**class** evalml.pipelines.**XGBoostRegressor**(*eta: float = 0.1*, *max_depth: int = 6*, *min_child_weight: int = 1*, *n_estimators: int = 100*, *random_seed: Union[int, float] = 0*, *n_jobs: int = 12*, ***kwargs*)

XGBoost Regressor.

**Parameters**

- **eta** (*float*) – Boosting learning rate. Defaults to 0.1.
- **max_depth** (*int*) – Maximum tree depth for base learners. Defaults to 6.
- **min_child_weight** (*float*) – Minimum sum of instance weight (hessian) needed in a child. Defaults to 1.0
- **n_estimators** (*int*) – Number of gradient boosted trees. Equivalent to number of boosting rounds. Defaults to 100.
- **random_seed** (*int*) – Seed for the random number generator. Defaults to 0.
- **n_jobs** (*int*) – Number of parallel threads used to run xgboost. Note that creating thread contention will significantly slow down the algorithm. Defaults to 12.

**Attributes**

| | |
|---|---|
| **hyper-parameter_ranges** | { "eta": Real(0.000001, 1), "max_depth": Integer(1, 20), "min_child_weight": Real(1, 10), "n_estimators": Integer(1, 1000),} |
| **model_family** | ModelFamily.XGBOOST |
| **modifies_features** | True |
| **modifies_target** | False |
| **name** | XGBoost Regressor |
| **SEED_MAX** | None |
| **SEED_MIN** | None |
| **supported_problem_types** | [ ProblemTypes.REGRESSION, ProblemTypes.TIME_SERIES_REGRESSION,] |
| **training_only** | False |

**Methods**

| | |
|---|---|
| *clone* | Constructs a new component with the same parameters and random state. |
| *default_parameters* | Returns the default parameters for this component. |
| *describe* | Describe a component and its parameters. |
| *feature_importance* | Feature importance of fitted XGBoost regressor. |
| *fit* | Fits XGBoost regressor component to data. |
| *get_prediction_intervals* | Find the prediction intervals using the fitted XGBoostRegressor. |
| *load* | Loads component at file path. |
| *needs_fitting* | Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances. |
| *parameters* | Returns the parameters which were used to initialize the component. |
| *predict* | Make predictions using fitted XGBoost regressor. |
| *predict_proba* | Make probability estimates for labels. |
| *save* | Saves component at file path. |
| *update_parameters* | Updates the parameter dictionary of the component. |

**clone**(*self*)

Constructs a new component with the same parameters and random state.

**Returns** A new instance of this component with identical parameters and random state.

**default_parameters**(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

**Returns** Default parameters for this component.

**Return type** dict

**describe**(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

**Parameters**

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description as dictionary in the format {"name": name, "parameters": parameters}

**Returns** Returns dictionary if return_dict is True, else None.

**Return type** None or dict

**property feature_importance**(*self*) → pandas.Series

Feature importance of fitted XGBoost regressor.

**fit**(*self, X: pandas.DataFrame, y: Optional[pandas.Series] = None*)

Fits XGBoost regressor component to data.

**Parameters**

- **X** (*pd.DataFrame*) – The input training data of shape [n_samples, n_features].
- **y** (*pd.Series, optional*) – The target training data of length [n_samples].

**Returns** self

**get_prediction_intervals**(*self, X: pandas.DataFrame, y: Optional[pandas.Series] = None, coverage: List[float] = None, predictions: pandas.Series = None*) → Dict[str, pandas.Series]

Find the prediction intervals using the fitted XGBoostRegressor.

**Parameters**

- **X** (*pd.DataFrame*) – Data of shape [n_samples, n_features].
- **y** (*pd.Series*) – Target data. Ignored.
- **coverage** (*List[float]*) – A list of floats between the values 0 and 1 that the upper and lower bounds of the prediction interval should be calculated for.
- **predictions** (*pd.Series*) – Optional list of predictions to use. If None, will generate predictions using *X*.

**Returns** Prediction intervals, keys are in the format {coverage}_lower or {coverage}_upper.

**Return type** dict

**static load**(*file_path*)

Loads component at file path.

**Parameters** **file_path** (*str*) – Location to load file.

**Returns** ComponentBase object

**needs_fitting**(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need to be fit or whose fit methods do nothing.

**Returns** True.

**property parameters**(*self*)

Returns the parameters which were used to initialize the component.

**predict**(*self*, *X: pandas.DataFrame*) → pandas.Series

Make predictions using fitted XGBoost regressor.

**Parameters** **X** (*pd.DataFrame*) – Data of shape [n_samples, n_features].

**Returns** Predicted values.

**Return type** pd.Series

**predict_proba**(*self*, *X: pandas.DataFrame*) → pandas.Series

Make probability estimates for labels.

**Parameters** **X** (*pd.DataFrame*) – Features.

**Returns** Probability estimates.

**Return type** pd.Series

**Raises** **MethodPropertyNotFoundError** – If estimator does not have a predict_proba method or a component_obj that implements predict_proba.

**save**(*self*, *file_path*, *pickle_protocol=cloudpickle.DEFAULT_PROTOCOL*)

Saves component at file path.

**Parameters**

- **file_path** (*str*) – Location to save file.
- **pickle_protocol** (*int*) – The pickle data stream format.

**update_parameters**(*self*, *update_dict*, *reset_fit=True*)

Updates the parameter dictionary of the component.

**Parameters**

- **update_dict** (*dict*) – A dict of parameters to update.
- **reset_fit** (*bool, optional*) – If True, will set *_is_fitted* to False.

## Preprocessing

Preprocessing utilities.

## Subpackages

## data_splitters

Data splitter classes.

### Submodules

### no_split

Empty Data Splitter class.

### Module Contents

### Classes Summary

| | |
|---|---|
| *NoSplit* | Does not split the training data into training and validation sets. |

### Contents

**class** evalml.preprocessing.data_splitters.no_split.**NoSplit**(*random_seed=0*)

Does not split the training data into training and validation sets.

All data is passed as the training set, test data is simply an array of *None*. To be used for future unsupervised learning, should not be used in any of the currently supported pipelines.

**Parameters** **random_seed** (*int*) – The seed to use for random sampling. Defaults to 0. Not used.

**Methods**

| | |
|---|---|
| *get_n_splits* | Return the number of splits of this object. |
| *is_cv* | Returns whether or not the data splitter is a cross-validation data splitter. |
| *split* | Divide the data into training and testing sets, where the testing set is empty. |

**static get_n_splits**()

Return the number of splits of this object.

**Returns** Always returns 0.

**Return type** int

**property is_cv**(*self*)

Returns whether or not the data splitter is a cross-validation data splitter.

**Returns** If the splitter is a cross-validation data splitter

**Return type** bool

**split**(*self*, *X*, *y=None*)

Divide the data into training and testing sets, where the testing set is empty.

**Parameters**

- **X** (*pd.DataFrame*) – Dataframe of points to split
- **y** (*pd.Series*) – Series of points to split

**Returns** Indices to split data into training and test set

**Return type** list

### sk_splitters

SKLearn data splitter wrapper classes.

### Module Contents

### Classes Summary

| | |
|---|---|
| *KFold* | Wrapper class for sklearn's KFold splitter. |
| *StratifiedKFold* | Wrapper class for sklearn's Stratified KFold splitter. |

### Contents

**class** evalml.preprocessing.data_splitters.sk_splitters.**KFold**(*n_splits=5*, *, *shuffle=False*, *random_state=None*)

Wrapper class for sklearn's KFold splitter.

**Methods**

| | |
|---|---|
| *get_n_splits* | Returns the number of splitting iterations in the cross-validator |
| *is_cv* | Returns whether or not the data splitter is a cross-validation data splitter. |
| *split* | Generate indices to split data into training and test set. |

**get_n_splits**(*self*, *X=None*, *y=None*, *groups=None*)

Returns the number of splitting iterations in the cross-validator

**Parameters**

- **X** (*object*) – Always ignored, exists for compatibility.
- **y** (*object*) – Always ignored, exists for compatibility.
- **groups** (*object*) – Always ignored, exists for compatibility.

**Returns** **n_splits** – Returns the number of splitting iterations in the cross-validator.

**Return type** int

**property is_cv**(*self*)

Returns whether or not the data splitter is a cross-validation data splitter.

**Returns** If the splitter is a cross-validation data splitter

**Return type** bool

**split**(*self*, *X*, *y=None*, *groups=None*)

Generate indices to split data into training and test set.

**Parameters**

- **X** (*array-like of shape (n_samples, n_features)*) – Training data, where *n_samples* is the number of samples and *n_features* is the number of features.
- **y** (*array-like of shape (n_samples,), default=None*) – The target variable for supervised learning problems.
- **groups** (*array-like of shape (n_samples,), default=None*) – Group labels for the samples used while splitting the dataset into train/test set.

**Yields**

- **train** (*ndarray*) – The training set indices for that split.
- **test** (*ndarray*) – The testing set indices for that split.

**class** evalml.preprocessing.data_splitters.sk_splitters.**StratifiedKFold**(*n_splits=5*, ***, *shuffle=False*, *random_state=None*)

Wrapper class for sklearn's Stratified KFold splitter.

**Methods**

| | |
|---|---|
| *get_n_splits* | Returns the number of splitting iterations in the cross-validator |
| *is_cv* | Returns whether or not the data splitter is a cross-validation data splitter. |
| *split* | Generate indices to split data into training and test set. |

**get_n_splits**(*self*, *X=None*, *y=None*, *groups=None*)

Returns the number of splitting iterations in the cross-validator

**Parameters**

- **X** (*object*) – Always ignored, exists for compatibility.
- **y** (*object*) – Always ignored, exists for compatibility.
- **groups** (*object*) – Always ignored, exists for compatibility.

**Returns** **n_splits** – Returns the number of splitting iterations in the cross-validator.

**Return type** int

**property is_cv**(*self*)

Returns whether or not the data splitter is a cross-validation data splitter.

**Returns** If the splitter is a cross-validation data splitter

**Return type** bool

**split**(*self*, *X*, *y*, *groups=None*)

Generate indices to split data into training and test set.

**Parameters**

- **X** (*array-like of shape (n_samples, n_features)*) – Training data, where *n_samples* is the number of samples and *n_features* is the number of features.

  Note that providing `y` is sufficient to generate the splits and hence `np.zeros(n_samples)` may be used as a placeholder for `X` instead of actual training data.

- **y** (*array-like of shape (n_samples,)*) – The target variable for supervised learning problems. Stratification is done based on the y labels.
- **groups** (*object*) – Always ignored, exists for compatibility.

**Yields**

- **train** (*ndarray*) – The training set indices for that split.
- **test** (*ndarray*) – The testing set indices for that split.

### Notes

Randomized CV splitters may return different results for each call of split. You can make the results identical by setting *random_state* to an integer.

## time_series_split

Rolling Origin Cross Validation for time series problems.

### Module Contents

### Classes Summary

| | |
|---|---|
| *TimeSeriesSplit* | Rolling Origin Cross Validation for time series problems. |

### Contents

**class** evalml.preprocessing.data_splitters.time_series_split.**TimeSeriesSplit**(*max_delay=0, gap=0, forecast_horizon=None, time_index=None, n_splits=3*)

Rolling Origin Cross Validation for time series problems.

The max_delay, gap, and forecast_horizon parameters are only used to validate that the requested split size is not too small given these parameters.

**Parameters**

- **max_delay** (*int*) – Max delay value for feature engineering. Time series pipelines create delayed features from existing features. This process will introduce NaNs into the first max_delay number of rows. The splitter uses the last max_delay number of rows from the previous split as the first max_delay number of rows of the current split to avoid "throwing out" more data than in necessary. Defaults to 0.
- **gap** (*int*) – Number of time units separating the data used to generate features and the data to forecast on. Defaults to 0.
- **forecast_horizon** (*int, None*) – Number of time units to forecast. Used for parameter validation. If an integer, will set the size of the cv splits. Defaults to None.

- **time_index** (*str*) – Name of the column containing the datetime information used to order the data. Defaults to None.
- **n_splits** (*int*) – number of data splits to make. Defaults to 3.

### Example

```
>>> import numpy as np
>>> import pandas as pd
...
>>> X = pd.DataFrame([i for i in range(10)], columns=["First"])
>>> y = pd.Series([i for i in range(10)])
...
>>> ts_split = TimeSeriesSplit(n_splits=4)
>>> generator_ = ts_split.split(X, y)
...
>>> first_split = next(generator_)
>>> assert (first_split[0] == np.array([0, 1])).all()
>>> assert (first_split[1] == np.array([2, 3])).all()
...
...
>>> second_split = next(generator_)
>>> assert (second_split[0] == np.array([0, 1, 2, 3])).all()
>>> assert (second_split[1] == np.array([4, 5])).all()
...
...
>>> third_split = next(generator_)
>>> assert (third_split[0] == np.array([0, 1, 2, 3, 4, 5])).all()
>>> assert (third_split[1] == np.array([6, 7])).all()
...
...
>>> fourth_split = next(generator_)
>>> assert (fourth_split[0] == np.array([0, 1, 2, 3, 4, 5, 6, 7])).all()
>>> assert (fourth_split[1] == np.array([8, 9])).all()
```

**Methods**

| | |
|---|---|
| *get_n_splits* | Get the number of data splits. |
| *is_cv* | Returns whether or not the data splitter is a cross-validation data splitter. |
| *split* | Get the time series splits. |

**get_n_splits**(*self*, *X=None*, *y=None*, *groups=None*)

Get the number of data splits.

**Parameters**

- **X** (*pd.DataFrame, None*) – Features to split.
- **y** (*pd.DataFrame, None*) – Target variable to split. Defaults to None.
- **groups** – Ignored but kept for compatibility with sklearn API. Defaults to None.

**Returns** Number of splits.

**property is_cv**(*self*)

Returns whether or not the data splitter is a cross-validation data splitter.

**Returns** If the splitter is a cross-validation data splitter

**Return type** bool

**split**(*self*, *X*, *y=None*, *groups=None*)

Get the time series splits.

X and y are assumed to be sorted in ascending time order. This method can handle passing in empty or None X and y data but note that X and y cannot be None or empty at the same time.

**Parameters**

- **X** (*pd.DataFrame, None*) – Features to split.
- **y** (*pd.DataFrame, None*) – Target variable to split. Defaults to None.
- **groups** – Ignored but kept for compatibility with sklearn API. Defaults to None.

**Yields** Iterator of (train, test) indices tuples.

**Raises** **ValueError** – If one of the proposed splits would be empty.

## training_validation_split

Training Validation Split class.

### Module Contents

### Classes Summary

| | |
|---|---|
| *TrainingValidationSplit* | Split the training data into training and validation sets. |

### Contents

**class** evalml.preprocessing.data_splitters.training_validation_split.**TrainingValidationSplit**(*test_size=None, train_size=Non, shuffle=False, stratify=None, random_seed=0*)

Split the training data into training and validation sets.

**Parameters**

- **test_size** (*float*) – What percentage of data points should be included in the validation set. Defalts to the complement of *train_size* if *train_size* is set, and 0.25 otherwise.
- **train_size** (*float*) – What percentage of data points should be included in the training set. Defaults to the complement of *test_size*
- **shuffle** (*boolean*) – Whether to shuffle the data before splitting. Defaults to False.

- **stratify** (*list*) – Splits the data in a stratified fashion, using this argument as class labels. Defaults to None.
- **random_seed** (*int*) – The seed to use for random sampling. Defaults to 0.

### Examples

```
>>> import numpy as np
>>> import pandas as pd
...
>>> X = pd.DataFrame([i for i in range(10)], columns=["First"])
>>> y = pd.Series([i for i in range(10)])
...
>>> tv_split = TrainingValidationSplit()
>>> split_ = next(tv_split.split(X, y))
>>> assert (split_[0] == np.array([0, 1, 2, 3, 4, 5, 6])).all()
>>> assert (split_[1] == np.array([7, 8, 9])).all()
...
...
>>> tv_split = TrainingValidationSplit(test_size=0.5)
>>> split_ = next(tv_split.split(X, y))
>>> assert (split_[0] == np.array([0, 1, 2, 3, 4])).all()
>>> assert (split_[1] == np.array([5, 6, 7, 8, 9])).all()
...
...
>>> tv_split = TrainingValidationSplit(shuffle=True)
>>> split_ = next(tv_split.split(X, y))
>>> assert (split_[0] == np.array([9, 1, 6, 7, 3, 0, 5])).all()
>>> assert (split_[1] == np.array([2, 8, 4])).all()
...
...
>>> y = pd.Series([i % 3 for i in range(10)])
>>> tv_split = TrainingValidationSplit(shuffle=True, stratify=y)
>>> split_ = next(tv_split.split(X, y))
>>> assert (split_[0] == np.array([1, 9, 3, 2, 8, 6, 7])).all()
>>> assert (split_[1] == np.array([0, 4, 5])).all()
```

#### Methods

| | |
|---|---|
| *get_n_splits* | Return the number of splits of this object. |
| *is_cv* | Returns whether or not the data splitter is a cross-validation data splitter. |
| *split* | Divide the data into training and testing sets. |

**static get_n_splits()**

Return the number of splits of this object.

**Returns** Always returns 1.

**Return type** int

**property is_cv(*self*)**

Returns whether or not the data splitter is a cross-validation data splitter.

**Returns** If the splitter is a cross-validation data splitter

**Return type** bool

**split**(*self*, *X*, *y=None*)

Divide the data into training and testing sets.

**Parameters**

- **X** (*pd.DataFrame*) – Dataframe of points to split
- **y** (*pd.Series*) – Series of points to split

**Returns** Indices to split data into training and test set

**Return type** list

## Package Contents

## Classes Summary

| | |
|---|---|
| *KFold* | Wrapper class for sklearn's KFold splitter. |
| *NoSplit* | Does not split the training data into training and validation sets. |
| *StratifiedKFold* | Wrapper class for sklearn's Stratified KFold splitter. |
| *TimeSeriesSplit* | Rolling Origin Cross Validation for time series problems. |
| *TrainingValidationSplit* | Split the training data into training and validation sets. |

## Contents

**class** evalml.preprocessing.data_splitters.**KFold**(*n_splits=5*, ***, *shuffle=False*, *random_state=None*)

Wrapper class for sklearn's KFold splitter.

**Methods**

| | |
|---|---|
| *get_n_splits* | Returns the number of splitting iterations in the cross-validator |
| *is_cv* | Returns whether or not the data splitter is a cross-validation data splitter. |
| *split* | Generate indices to split data into training and test set. |

**get_n_splits**(*self*, *X=None*, *y=None*, *groups=None*)

Returns the number of splitting iterations in the cross-validator

**Parameters**

- **X** (*object*) – Always ignored, exists for compatibility.
- **y** (*object*) – Always ignored, exists for compatibility.
- **groups** (*object*) – Always ignored, exists for compatibility.

**Returns** **n_splits** – Returns the number of splitting iterations in the cross-validator.

**Return type** int

**property is_cv**(*self*)

Returns whether or not the data splitter is a cross-validation data splitter.

**Returns** If the splitter is a cross-validation data splitter

**Return type** bool

**split**(*self*, *X*, *y=None*, *groups=None*)

Generate indices to split data into training and test set.

**Parameters**

- **X** (*array-like of shape (n_samples, n_features)*) – Training data, where *n_samples* is the number of samples and *n_features* is the number of features.
- **y** (*array-like of shape (n_samples,), default=None*) – The target variable for supervised learning problems.
- **groups** (*array-like of shape (n_samples,), default=None*) – Group labels for the samples used while splitting the dataset into train/test set.

**Yields**

- **train** (*ndarray*) – The training set indices for that split.
- **test** (*ndarray*) – The testing set indices for that split.

**class** evalml.preprocessing.data_splitters.**NoSplit**(*random_seed=0*)

Does not split the training data into training and validation sets.

All data is passed as the training set, test data is simply an array of *None*. To be used for future unsupervised learning, should not be used in any of the currently supported pipelines.

**Parameters** **random_seed** (*int*) – The seed to use for random sampling. Defaults to 0. Not used.

**Methods**

| | |
|---|---|
| *get_n_splits* | Return the number of splits of this object. |
| *is_cv* | Returns whether or not the data splitter is a cross-validation data splitter. |
| *split* | Divide the data into training and testing sets, where the testing set is empty. |

**static get_n_splits**()

Return the number of splits of this object.

**Returns** Always returns 0.

**Return type** int

**property is_cv**(*self*)

Returns whether or not the data splitter is a cross-validation data splitter.

**Returns** If the splitter is a cross-validation data splitter

**Return type** bool

**split**(*self*, *X*, *y=None*)

Divide the data into training and testing sets, where the testing set is empty.

**Parameters**

- **X** (*pd.DataFrame*) – Dataframe of points to split

- **y** (*pd.Series*) – Series of points to split

**Returns** Indices to split data into training and test set

**Return type** list

**class** evalml.preprocessing.data_splitters.**StratifiedKFold**(*n_splits=5*, ***, *shuffle=False*, *random_state=None*)

Wrapper class for sklearn's Stratified KFold splitter.

**Methods**

| | |
|---|---|
| *get_n_splits* | Returns the number of splitting iterations in the cross-validator |
| *is_cv* | Returns whether or not the data splitter is a cross-validation data splitter. |
| *split* | Generate indices to split data into training and test set. |

**get_n_splits**(*self*, *X=None*, *y=None*, *groups=None*)

Returns the number of splitting iterations in the cross-validator

**Parameters**

- **X** (*object*) – Always ignored, exists for compatibility.
- **y** (*object*) – Always ignored, exists for compatibility.
- **groups** (*object*) – Always ignored, exists for compatibility.

**Returns** **n_splits** – Returns the number of splitting iterations in the cross-validator.

**Return type** int

**property is_cv**(*self*)

Returns whether or not the data splitter is a cross-validation data splitter.

**Returns** If the splitter is a cross-validation data splitter

**Return type** bool

**split**(*self*, *X*, *y*, *groups=None*)

Generate indices to split data into training and test set.

**Parameters**

- **X** (*array-like of shape (n_samples, n_features)*) – Training data, where *n_samples* is the number of samples and *n_features* is the number of features.

  Note that providing `y` is sufficient to generate the splits and hence `np.zeros(n_samples)` may be used as a placeholder for `X` instead of actual training data.
- **y** (*array-like of shape (n_samples,)*) – The target variable for supervised learning problems. Stratification is done based on the y labels.
- **groups** (*object*) – Always ignored, exists for compatibility.

**Yields**

- **train** (*ndarray*) – The training set indices for that split.
- **test** (*ndarray*) – The testing set indices for that split.

### Notes

Randomized CV splitters may return different results for each call of split. You can make the results identical by setting *random_state* to an integer.

**class** evalml.preprocessing.data_splitters.**TimeSeriesSplit**(*max_delay=0*, *gap=0*, *forecast_horizon=None*, *time_index=None*, *n_splits=3*)

Rolling Origin Cross Validation for time series problems.

The max_delay, gap, and forecast_horizon parameters are only used to validate that the requested split size is not too small given these parameters.

**Parameters**

- **max_delay** (*int*) – Max delay value for feature engineering. Time series pipelines create delayed features from existing features. This process will introduce NaNs into the first max_delay number of rows. The splitter uses the last max_delay number of rows from the previous split as the first max_delay number of rows of the current split to avoid "throwing out" more data than in necessary. Defaults to 0.
- **gap** (*int*) – Number of time units separating the data used to generate features and the data to forecast on. Defaults to 0.
- **forecast_horizon** (*int, None*) – Number of time units to forecast. Used for parameter validation. If an integer, will set the size of the cv splits. Defaults to None.
- **time_index** (*str*) – Name of the column containing the datetime information used to order the data. Defaults to None.
- **n_splits** (*int*) – number of data splits to make. Defaults to 3.

### Example

```
>>> import numpy as np
>>> import pandas as pd
...
>>> X = pd.DataFrame([i for i in range(10)], columns=["First"])
>>> y = pd.Series([i for i in range(10)])
...
>>> ts_split = TimeSeriesSplit(n_splits=4)
>>> generator_ = ts_split.split(X, y)
...
>>> first_split = next(generator_)
>>> assert (first_split[0] == np.array([0, 1])).all()
>>> assert (first_split[1] == np.array([2, 3])).all()
...
...
>>> second_split = next(generator_)
>>> assert (second_split[0] == np.array([0, 1, 2, 3])).all()
>>> assert (second_split[1] == np.array([4, 5])).all()
...
...
>>> third_split = next(generator_)
>>> assert (third_split[0] == np.array([0, 1, 2, 3, 4, 5])).all()
>>> assert (third_split[1] == np.array([6, 7])).all()
```

(continues on next page)

(continued from previous page)

```
...
...
>>> fourth_split = next(generator_)
>>> assert (fourth_split[0] == np.array([0, 1, 2, 3, 4, 5, 6, 7])).all()
>>> assert (fourth_split[1] == np.array([8, 9])).all()
```

**Methods**

| | |
|---|---|
| *get_n_splits* | Get the number of data splits. |
| *is_cv* | Returns whether or not the data splitter is a cross-validation data splitter. |
| *split* | Get the time series splits. |

**get_n_splits**(*self*, *X=None*, *y=None*, *groups=None*)

Get the number of data splits.

**Parameters**

- **X** (*pd.DataFrame, None*) – Features to split.
- **y** (*pd.DataFrame, None*) – Target variable to split. Defaults to None.
- **groups** – Ignored but kept for compatibility with sklearn API. Defaults to None.

**Returns** Number of splits.

**property is_cv**(*self*)

Returns whether or not the data splitter is a cross-validation data splitter.

**Returns** If the splitter is a cross-validation data splitter

**Return type** bool

**split**(*self*, *X*, *y=None*, *groups=None*)

Get the time series splits.

X and y are assumed to be sorted in ascending time order. This method can handle passing in empty or None X and y data but note that X and y cannot be None or empty at the same time.

**Parameters**

- **X** (*pd.DataFrame, None*) – Features to split.
- **y** (*pd.DataFrame, None*) – Target variable to split. Defaults to None.
- **groups** – Ignored but kept for compatibility with sklearn API. Defaults to None.

**Yields** Iterator of (train, test) indices tuples.

**Raises** **ValueError** – If one of the proposed splits would be empty.

**class** evalml.preprocessing.data_splitters.**TrainingValidationSplit**(*test_size=None*, *train_size=None*, *shuffle=False*, *stratify=None*, *random_seed=0*)

Split the training data into training and validation sets.

**Parameters**

- **test_size** (*float*) – What percentage of data points should be included in the validation set. Defalts to the complement of *train_size* if *train_size* is set, and 0.25 otherwise.

- **train_size** (*float*) – What percentage of data points should be included in the training set. Defaults to the complement of *test_size*
- **shuffle** (*boolean*) – Whether to shuffle the data before splitting. Defaults to False.
- **stratify** (*list*) – Splits the data in a stratified fashion, using this argument as class labels. Defaults to None.
- **random_seed** (*int*) – The seed to use for random sampling. Defaults to 0.

### Examples

```
>>> import numpy as np
>>> import pandas as pd
...
>>> X = pd.DataFrame([i for i in range(10)], columns=["First"])
>>> y = pd.Series([i for i in range(10)])
...
>>> tv_split = TrainingValidationSplit()
>>> split_ = next(tv_split.split(X, y))
>>> assert (split_[0] == np.array([0, 1, 2, 3, 4, 5, 6])).all()
>>> assert (split_[1] == np.array([7, 8, 9])).all()
...
...
>>> tv_split = TrainingValidationSplit(test_size=0.5)
>>> split_ = next(tv_split.split(X, y))
>>> assert (split_[0] == np.array([0, 1, 2, 3, 4])).all()
>>> assert (split_[1] == np.array([5, 6, 7, 8, 9])).all()
...
...
>>> tv_split = TrainingValidationSplit(shuffle=True)
>>> split_ = next(tv_split.split(X, y))
>>> assert (split_[0] == np.array([9, 1, 6, 7, 3, 0, 5])).all()
>>> assert (split_[1] == np.array([2, 8, 4])).all()
...
...
>>> y = pd.Series([i % 3 for i in range(10)])
>>> tv_split = TrainingValidationSplit(shuffle=True, stratify=y)
>>> split_ = next(tv_split.split(X, y))
>>> assert (split_[0] == np.array([1, 9, 3, 2, 8, 6, 7])).all()
>>> assert (split_[1] == np.array([0, 4, 5])).all()
```

#### Methods

| | |
|---|---|
| *get_n_splits* | Return the number of splits of this object. |
| *is_cv* | Returns whether or not the data splitter is a cross-validation data splitter. |
| *split* | Divide the data into training and testing sets. |

**static get_n_splits()**

Return the number of splits of this object.

**Returns** Always returns 1.

**Return type** int

**property is_cv**(*self*)

Returns whether or not the data splitter is a cross-validation data splitter.

**Returns** If the splitter is a cross-validation data splitter

**Return type** bool

**split**(*self*, *X*, *y=None*)

Divide the data into training and testing sets.

**Parameters**

- **X** (*pd.DataFrame*) – Dataframe of points to split
- **y** (*pd.Series*) – Series of points to split

**Returns** Indices to split data into training and test set

**Return type** list

## Submodules

### utils

Helpful preprocessing utilities.

### Module Contents

### Functions

| | |
|---|---|
| *load_data* | Load features and target from file. |
| *number_of_features* | Get the number of features of each specific dtype in a DataFrame. |
| *split_data* | Split data into train and test sets. |
| *target_distribution* | Get the target distributions. |

### Contents

evalml.preprocessing.utils.**load_data**(*path*, *index*, *target*, *n_rows=None*, *drop=None*, *verbose=True*, ***kwargs*)

Load features and target from file.

**Parameters**

- **path** (*str*) – Path to file or a http/ftp/s3 URL.
- **index** (*str*) – Column for index.
- **target** (*str*) – Column for target.
- **n_rows** (*int*) – Number of rows to return. Defaults to None.
- **drop** (*list*) – List of columns to drop. Defaults to None.
- **verbose** (*bool*) – If True, prints information about features and target. Defaults to True.

- ****kwargs** – Other keyword arguments that should be passed to panda's *read_csv* method.

**Returns** Features matrix and target.

**Return type** pd.DataFrame, pd.Series

evalml.preprocessing.utils.**number_of_features**(*dtypes*)

Get the number of features of each specific dtype in a DataFrame.

**Parameters** **dtypes** (*pd.Series*) – DataFrame.dtypes to get the number of features for.

**Returns** dtypes and the number of features for each input type.

**Return type** pd.Series

### Example

```
>>> X = pd.DataFrame()
>>> X["integers"] = [i for i in range(10)]
>>> X["floats"] = [float(i) for i in range(10)]
>>> X["strings"] = [str(i) for i in range(10)]
>>> X["booleans"] = [bool(i%2) for i in range(10)]
```

Lists the number of columns corresponding to each dtype.

```
>>> number_of_features(X.dtypes)
             Number of Features
Boolean                       1
Categorical                   1
Numeric                       2
```

evalml.preprocessing.utils.**split_data**(*X*, *y*, *problem_type*, *problem_configuration=None*, *test_size=None*, *random_seed=0*)

Split data into train and test sets.

**Parameters**

- **X** (*pd.DataFrame or np.ndarray*) – data of shape [n_samples, n_features]
- **y** (*pd.Series, or np.ndarray*) – target data of length [n_samples]
- **problem_type** (*str or ProblemTypes*) – type of supervised learning problem. see evalml.problem_types.problemtype.all_problem_types for a full list.
- **problem_configuration** (*dict*) – Additional parameters needed to configure the search. For example, in time series problems, values should be passed in for the time_index, gap, and max_delay variables.
- **test_size** (*float*) – What percentage of data points should be included in the test set. Defaults to 0.2 (20%) for non-timeseries problems and 0.1 (10%) for timeseries problems.
- **random_seed** (*int*) – Seed for the random number generator. Defaults to 0.

**Returns** Feature and target data each split into train and test sets.

**Return type** pd.DataFrame, pd.DataFrame, pd.Series, pd.Series

### Examples

```
>>> X = pd.DataFrame([1, 2, 3, 4, 5, 6], columns=["First"])
>>> y = pd.Series([8, 9, 10, 11, 12, 13])
...
>>> X_train, X_validation, y_train, y_validation = split_data(X, y, "regression",␣
↪random_seed=42)
>>> X_train
   First
5      6
2      3
4      5
3      4
>>> X_validation
   First
0      1
1      2
>>> y_train
5    13
2    10
4    12
3    11
dtype: int64
>>> y_validation
0    8
1    9
dtype: int64
```

evalml.preprocessing.utils.**target_distribution**(*targets*)

Get the target distributions.

**Parameters** **targets** (*pd.Series*) – Target data.

**Returns** Target data and their frequency distribution as percentages.

**Return type** pd.Series

### Examples

```
>>> y = pd.Series([1, 2, 4, 1, 3, 3, 1, 2])
>>> target_distribution(y)
Targets
1    37.50%
2    25.00%
3    25.00%
4    12.50%
dtype: object
>>> y = pd.Series([True, False, False, False, True])
>>> target_distribution(y)
Targets
False    60.00%
True     40.00%
dtype: object
```

### Package Contents

### Classes Summary

| | |
|---|---|
| *NoSplit* | Does not split the training data into training and validation sets. |
| *TimeSeriesSplit* | Rolling Origin Cross Validation for time series problems. |
| *TrainingValidationSplit* | Split the training data into training and validation sets. |

### Functions

| | |
|---|---|
| *load_data* | Load features and target from file. |
| *number_of_features* | Get the number of features of each specific dtype in a DataFrame. |
| *split_data* | Split data into train and test sets. |
| *target_distribution* | Get the target distributions. |

### Contents

`evalml.preprocessing.load_data(path, index, target, n_rows=None, drop=None, verbose=True, **kwargs)`

Load features and target from file.

**Parameters**

- **path** (*str*) – Path to file or a http/ftp/s3 URL.
- **index** (*str*) – Column for index.
- **target** (*str*) – Column for target.
- **n_rows** (*int*) – Number of rows to return. Defaults to None.
- **drop** (*list*) – List of columns to drop. Defaults to None.
- **verbose** (*bool*) – If True, prints information about features and target. Defaults to True.
- ****kwargs** – Other keyword arguments that should be passed to panda's *read_csv* method.

**Returns** Features matrix and target.

**Return type** pd.DataFrame, pd.Series

`class evalml.preprocessing.NoSplit(random_seed=0)`

Does not split the training data into training and validation sets.

All data is passed as the training set, test data is simply an array of *None*. To be used for future unsupervised learning, should not be used in any of the currently supported pipelines.

**Parameters** **random_seed** (*int*) – The seed to use for random sampling. Defaults to 0. Not used.

**Methods**

| | |
|---|---|
| *get_n_splits* | Return the number of splits of this object. |
| *is_cv* | Returns whether or not the data splitter is a cross-validation data splitter. |
| *split* | Divide the data into training and testing sets, where the testing set is empty. |

**static get_n_splits()**

Return the number of splits of this object.

**Returns** Always returns 0.

**Return type** int

**property is_cv**(*self*)

Returns whether or not the data splitter is a cross-validation data splitter.

**Returns** If the splitter is a cross-validation data splitter

**Return type** bool

**split**(*self*, *X*, *y=None*)

Divide the data into training and testing sets, where the testing set is empty.

**Parameters**

- **X** (*pd.DataFrame*) – Dataframe of points to split
- **y** (*pd.Series*) – Series of points to split

**Returns** Indices to split data into training and test set

**Return type** list

evalml.preprocessing.**number_of_features**(*dtypes*)

Get the number of features of each specific dtype in a DataFrame.

**Parameters** **dtypes** (*pd.Series*) – DataFrame.dtypes to get the number of features for.

**Returns** dtypes and the number of features for each input type.

**Return type** pd.Series

### Example

```
>>> X = pd.DataFrame()
>>> X["integers"] = [i for i in range(10)]
>>> X["floats"] = [float(i) for i in range(10)]
>>> X["strings"] = [str(i) for i in range(10)]
>>> X["booleans"] = [bool(i%2) for i in range(10)]
```

Lists the number of columns corresponding to each dtype.

```
>>> number_of_features(X.dtypes)
             Number of Features
Boolean                       1
Categorical                   1
Numeric                       2
```

evalml.preprocessing.**split_data**(*X*, *y*, *problem_type*, *problem_configuration=None*, *test_size=None*, *random_seed=0*)

Split data into train and test sets.

**Parameters**

- **X** (*pd.DataFrame or np.ndarray*) – data of shape [n_samples, n_features]
- **y** (*pd.Series, or np.ndarray*) – target data of length [n_samples]
- **problem_type** (*str or ProblemTypes*) – type of supervised learning problem. see evalml.problem_types.problemtype.all_problem_types for a full list.
- **problem_configuration** (*dict*) – Additional parameters needed to configure the search. For example, in time series problems, values should be passed in for the time_index, gap, and max_delay variables.
- **test_size** (*float*) – What percentage of data points should be included in the test set. Defaults to 0.2 (20%) for non-timeseries problems and 0.1 (10%) for timeseries problems.
- **random_seed** (*int*) – Seed for the random number generator. Defaults to 0.

**Returns** Feature and target data each split into train and test sets.

**Return type** pd.DataFrame, pd.DataFrame, pd.Series, pd.Series

### Examples

```
>>> X = pd.DataFrame([1, 2, 3, 4, 5, 6], columns=["First"])
>>> y = pd.Series([8, 9, 10, 11, 12, 13])
...
>>> X_train, X_validation, y_train, y_validation = split_data(X, y, "regression",␣
↪random_seed=42)
>>> X_train
   First
5      6
2      3
4      5
3      4
>>> X_validation
   First
0      1
1      2
>>> y_train
5    13
2    10
4    12
3    11
dtype: int64
>>> y_validation
0    8
1    9
dtype: int64
```

evalml.preprocessing.**target_distribution**(*targets*)

Get the target distributions.

**Parameters** **targets** (*pd.Series*) – Target data.

**Returns** Target data and their frequency distribution as percentages.

**Return type** pd.Series

### Examples

```
>>> y = pd.Series([1, 2, 4, 1, 3, 3, 1, 2])
>>> target_distribution(y)
Targets
1    37.50%
2    25.00%
3    25.00%
4    12.50%
dtype: object
>>> y = pd.Series([True, False, False, False, True])
>>> target_distribution(y)
Targets
False    60.00%
True     40.00%
dtype: object
```

**class** evalml.preprocessing.**TimeSeriesSplit**(*max_delay=0*, *gap=0*, *forecast_horizon=None*, *time_index=None*, *n_splits=3*)

Rolling Origin Cross Validation for time series problems.

The max_delay, gap, and forecast_horizon parameters are only used to validate that the requested split size is not too small given these parameters.

**Parameters**

- **max_delay** (*int*) – Max delay value for feature engineering. Time series pipelines create delayed features from existing features. This process will introduce NaNs into the first max_delay number of rows. The splitter uses the last max_delay number of rows from the previous split as the first max_delay number of rows of the current split to avoid "throwing out" more data than in necessary. Defaults to 0.
- **gap** (*int*) – Number of time units separating the data used to generate features and the data to forecast on. Defaults to 0.
- **forecast_horizon** (*int, None*) – Number of time units to forecast. Used for parameter validation. If an integer, will set the size of the cv splits. Defaults to None.
- **time_index** (*str*) – Name of the column containing the datetime information used to order the data. Defaults to None.
- **n_splits** (*int*) – number of data splits to make. Defaults to 3.

### Example

```
>>> import numpy as np
>>> import pandas as pd
...
>>> X = pd.DataFrame([i for i in range(10)], columns=["First"])
>>> y = pd.Series([i for i in range(10)])
...
>>> ts_split = TimeSeriesSplit(n_splits=4)
>>> generator_ = ts_split.split(X, y)
...
>>> first_split = next(generator_)
>>> assert (first_split[0] == np.array([0, 1])).all()
>>> assert (first_split[1] == np.array([2, 3])).all()
...
...
>>> second_split = next(generator_)
>>> assert (second_split[0] == np.array([0, 1, 2, 3])).all()
>>> assert (second_split[1] == np.array([4, 5])).all()
...
...
>>> third_split = next(generator_)
>>> assert (third_split[0] == np.array([0, 1, 2, 3, 4, 5])).all()
>>> assert (third_split[1] == np.array([6, 7])).all()
...
...
>>> fourth_split = next(generator_)
>>> assert (fourth_split[0] == np.array([0, 1, 2, 3, 4, 5, 6, 7])).all()
>>> assert (fourth_split[1] == np.array([8, 9])).all()
```

**Methods**

| | |
|---|---|
| *get_n_splits* | Get the number of data splits. |
| *is_cv* | Returns whether or not the data splitter is a cross-validation data splitter. |
| *split* | Get the time series splits. |

**get_n_splits**(*self*, *X=None*, *y=None*, *groups=None*)

Get the number of data splits.

**Parameters**

- **X** (*pd.DataFrame, None*) – Features to split.
- **y** (*pd.DataFrame, None*) – Target variable to split. Defaults to None.
- **groups** – Ignored but kept for compatibility with sklearn API. Defaults to None.

**Returns** Number of splits.

**property is_cv**(*self*)

Returns whether or not the data splitter is a cross-validation data splitter.

**Returns** If the splitter is a cross-validation data splitter

**Return type** bool

**split**(*self*, *X*, *y=None*, *groups=None*)

Get the time series splits.

X and y are assumed to be sorted in ascending time order. This method can handle passing in empty or None X and y data but note that X and y cannot be None or empty at the same time.

**Parameters**

- **X** (*pd.DataFrame, None*) – Features to split.
- **y** (*pd.DataFrame, None*) – Target variable to split. Defaults to None.
- **groups** – Ignored but kept for compatibility with sklearn API. Defaults to None.

**Yields** Iterator of (train, test) indices tuples.

**Raises** **ValueError** – If one of the proposed splits would be empty.

**class** evalml.preprocessing.**TrainingValidationSplit**(*test_size=None*, *train_size=None*, *shuffle=False*, *stratify=None*, *random_seed=0*)

Split the training data into training and validation sets.

**Parameters**

- **test_size** (*float*) – What percentage of data points should be included in the validation set. Defalts to the complement of *train_size* if *train_size* is set, and 0.25 otherwise.
- **train_size** (*float*) – What percentage of data points should be included in the training set. Defaults to the complement of *test_size*
- **shuffle** (*boolean*) – Whether to shuffle the data before splitting. Defaults to False.
- **stratify** (*list*) – Splits the data in a stratified fashion, using this argument as class labels. Defaults to None.
- **random_seed** (*int*) – The seed to use for random sampling. Defaults to 0.

### Examples

```
>>> import numpy as np
>>> import pandas as pd
...
>>> X = pd.DataFrame([i for i in range(10)], columns=["First"])
>>> y = pd.Series([i for i in range(10)])
...
>>> tv_split = TrainingValidationSplit()
>>> split_ = next(tv_split.split(X, y))
>>> assert (split_[0] == np.array([0, 1, 2, 3, 4, 5, 6])).all()
>>> assert (split_[1] == np.array([7, 8, 9])).all()
...
...
>>> tv_split = TrainingValidationSplit(test_size=0.5)
>>> split_ = next(tv_split.split(X, y))
>>> assert (split_[0] == np.array([0, 1, 2, 3, 4])).all()
>>> assert (split_[1] == np.array([5, 6, 7, 8, 9])).all()
...
...
>>> tv_split = TrainingValidationSplit(shuffle=True)
>>> split_ = next(tv_split.split(X, y))
```

(continues on next page)

(continued from previous page)

```
>>> assert (split_[0] == np.array([9, 1, 6, 7, 3, 0, 5])).all()
>>> assert (split_[1] == np.array([2, 8, 4])).all()
...
...
>>> y = pd.Series([i % 3 for i in range(10)])
>>> tv_split = TrainingValidationSplit(shuffle=True, stratify=y)
>>> split_ = next(tv_split.split(X, y))
>>> assert (split_[0] == np.array([1, 9, 3, 2, 8, 6, 7])).all()
>>> assert (split_[1] == np.array([0, 4, 5])).all()
```

**Methods**

| | |
|---|---|
| *get_n_splits* | Return the number of splits of this object. |
| *is_cv* | Returns whether or not the data splitter is a cross-validation data splitter. |
| *split* | Divide the data into training and testing sets. |

**static get_n_splits()**

Return the number of splits of this object.

**Returns** Always returns 1.

**Return type** int

**property is_cv(*self*)**

Returns whether or not the data splitter is a cross-validation data splitter.

**Returns** If the splitter is a cross-validation data splitter

**Return type** bool

**split(*self*, *X*, *y=None*)**

Divide the data into training and testing sets.

**Parameters**

- **X** (*pd.DataFrame*) – Dataframe of points to split
- **y** (*pd.Series*) – Series of points to split

**Returns** Indices to split data into training and test set

**Return type** list

## Problem Types

The supported types of machine learning problems.

## Submodules

### problem_types

Enum defining the supported types of machine learning problems.

## Module Contents

## Classes Summary

| | |
|---|---|
| *ProblemTypes* | Enum defining the supported types of machine learning problems. |

## Contents

**class** evalml.problem_types.problem_types.**ProblemTypes**

Enum defining the supported types of machine learning problems.

**Attributes**

| | |
|---|---|
| **BINARY** | Binary classification problem. |
| **MULTICLASS** | Multiclass classification problem. |
| **REGRESSION** | Regression problem. |
| **TIME_SERIES_BINARY** | Time series binary classification problem. |
| **TIME_SERIES_MULTICLASS** | Time series multiclass classification problem. |
| **TIME_SERIES_REGRESSION** | Time series regression problem. |

**Methods**

| | |
|---|---|
| *all_problem_types* | Get a list of all defined problem types. |
| *name* | The name of the Enum member. |
| *value* | The value of the Enum member. |

**all_problem_types**(*cls*)

Get a list of all defined problem types.

**Returns** List of all defined problem types.

**Return type** list(*ProblemTypes*)

**name**(*self*)

The name of the Enum member.

**value**(*self*)

The value of the Enum member.

### utils

Utility methods for the ProblemTypes enum in EvalML.

### Module Contents

### Functions

| | |
|---|---|
| *detect_problem_type* | Determine the type of problem is being solved based on the targets (binary vs multiclass classification, regression). Ignores missing and null data. |
| *handle_problem_types* | Handles problem_type by either returning the ProblemTypes or converting from a str. |
| *is_binary* | Determines if the provided problem_type is a binary classification problem type. |
| *is_classification* | Determines if the provided problem_type is a classification problem type. |
| *is_multiclass* | Determines if the provided problem_type is a multiclass classification problem type. |
| *is_regression* | Determines if the provided problem_type is a regression problem type. |
| *is_time_series* | Determines if the provided problem_type is a time series problem type. |

### Contents

evalml.problem_types.utils.**detect_problem_type**(*y*)

Determine the type of problem is being solved based on the targets (binary vs multiclass classification, regression). Ignores missing and null data.

**Parameters** **y** (*pd.Series*) – The target labels to predict.

**Returns** ProblemType Enum

**Return type** ProblemType

#### Examples

```
>>> y = pd.Series([0, 1, 0, 0, 0, 1, 1, 0, 0, 1, 1])
>>> assert detect_problem_type(y) == ProblemTypes.BINARY
...
>>> y = pd.Series([1, 2, 3, 2, 1, 1, 1, 2, 2, 3, 3])
>>> assert detect_problem_type(y) == ProblemTypes.MULTICLASS
...
>>> y = pd.Series([1.6, 4.2, 3.3, 2.9, 4, 1, 5.5, 2, -2, -3.2, 3])
>>> assert detect_problem_type(y) == ProblemTypes.REGRESSION
```

**Raises** **ValueError** – If the input has less than two classes.

evalml.problem_types.utils.**handle_problem_types**(*problem_type*)

Handles problem_type by either returning the ProblemTypes or converting from a str.

**Parameters** **problem_type** (*str or ProblemTypes*) – Problem type that needs to be handled.

**Returns** ProblemTypes enum

**Raises**

- **KeyError** – If input is not a valid ProblemTypes enum value.
- **ValueError** – If input is not a string or ProblemTypes object.

### Examples

```
>>> assert handle_problem_types("regression") == ProblemTypes.REGRESSION
>>> assert handle_problem_types("TIME SERIES BINARY") == ProblemTypes.TIME_SERIES_
↪BINARY
>>> assert handle_problem_types("Multiclass") == ProblemTypes.MULTICLASS
```

evalml.problem_types.utils.**is_binary**(*problem_type*)

Determines if the provided problem_type is a binary classification problem type.

**Parameters** **problem_type** (*str or ProblemTypes*) – type of supervised learning problem. See evalml.problem_types.ProblemType.all_problem_types for a full list.

**Returns** Whether or not the provided problem_type is a binary classification problem type.

**Return type** bool

### Examples

```
>>> assert is_binary("Binary")
>>> assert is_binary(ProblemTypes.BINARY)
>>> assert is_binary(ProblemTypes.TIME_SERIES_BINARY)
```

evalml.problem_types.utils.**is_classification**(*problem_type*)

Determines if the provided problem_type is a classification problem type.

**Parameters** **problem_type** (*str or ProblemTypes*) – type of supervised learning problem. See evalml.problem_types.ProblemType.all_problem_types for a full list.

**Returns** Whether or not the provided problem_type is a classification problem type.

**Return type** bool

### Examples

```
>>> assert is_classification("Multiclass")
>>> assert is_classification(ProblemTypes.TIME_SERIES_BINARY)
>>> assert not is_classification(ProblemTypes.REGRESSION)
```

evalml.problem_types.utils.**is_multiclass**(*problem_type*)

Determines if the provided problem_type is a multiclass classification problem type.

**Parameters** **problem_type** (*str or ProblemTypes*) – type of supervised learning problem. See evalml.problem_types.ProblemType.all_problem_types for a full list.

**Returns** Whether or not the provided problem_type is a multiclass classification problem type.

**Return type** bool

### Examples

```
>>> assert is_multiclass("Multiclass")
>>> assert is_multiclass(ProblemTypes.MULTICLASS)
>>> assert is_multiclass(ProblemTypes.TIME_SERIES_MULTICLASS)
```

evalml.problem_types.utils.**is_regression**(*problem_type*)

Determines if the provided problem_type is a regression problem type.

**Parameters** **problem_type** (*str or ProblemTypes*) – type of supervised learning problem. See evalml.problem_types.ProblemType.all_problem_types for a full list.

**Returns** Whether or not the provided problem_type is a regression problem type.

**Return type** bool

### Examples

```
>>> assert is_regression("Regression")
>>> assert is_regression(ProblemTypes.REGRESSION)
>>> assert is_regression(ProblemTypes.TIME_SERIES_REGRESSION)
```

evalml.problem_types.utils.**is_time_series**(*problem_type*)

Determines if the provided problem_type is a time series problem type.

**Parameters** **problem_type** (*str or ProblemTypes*) – type of supervised learning problem. See evalml.problem_types.ProblemType.all_problem_types for a full list.

**Returns** Whether or not the provided problem_type is a time series problem type.

**Return type** bool

### Examples

```
>>> assert is_time_series("time series regression")
>>> assert is_time_series(ProblemTypes.TIME_SERIES_BINARY)
>>> assert not is_time_series(ProblemTypes.REGRESSION)
```

### Package Contents

### Classes Summary

| | |
|---|---|
| *ProblemTypes* | Enum defining the supported types of machine learning problems. |

### Functions

| | |
|---|---|
| *detect_problem_type* | Determine the type of problem is being solved based on the targets (binary vs multiclass classification, regression). Ignores missing and null data. |
| *handle_problem_types* | Handles problem_type by either returning the ProblemTypes or converting from a str. |
| *is_binary* | Determines if the provided problem_type is a binary classification problem type. |
| *is_classification* | Determines if the provided problem_type is a classification problem type. |
| *is_multiclass* | Determines if the provided problem_type is a multiclass classification problem type. |
| *is_regression* | Determines if the provided problem_type is a regression problem type. |
| *is_time_series* | Determines if the provided problem_type is a time series problem type. |

### Contents

evalml.problem_types.**detect_problem_type**(*y*)

Determine the type of problem is being solved based on the targets (binary vs multiclass classification, regression). Ignores missing and null data.

**Parameters** **y** (*pd.Series*) – The target labels to predict.

**Returns** ProblemType Enum

**Return type** ProblemType

#### Examples

```
>>> y = pd.Series([0, 1, 0, 0, 0, 1, 1, 0, 0, 1, 1])
>>> assert detect_problem_type(y) == ProblemTypes.BINARY
...
>>> y = pd.Series([1, 2, 3, 2, 1, 1, 1, 2, 2, 3, 3])
>>> assert detect_problem_type(y) == ProblemTypes.MULTICLASS
...
>>> y = pd.Series([1.6, 4.2, 3.3, 2.9, 4, 1, 5.5, 2, -2, -3.2, 3])
>>> assert detect_problem_type(y) == ProblemTypes.REGRESSION
```

**Raises** **ValueError** – If the input has less than two classes.

evalml.problem_types.**handle_problem_types**(*problem_type*)

Handles problem_type by either returning the ProblemTypes or converting from a str.

**Parameters** **problem_type** (*str or* ProblemTypes) – Problem type that needs to be handled.

**Returns** ProblemTypes enum

**Raises**

- **KeyError** – If input is not a valid ProblemTypes enum value.
- **ValueError** – If input is not a string or ProblemTypes object.

### Examples

```
>>> assert handle_problem_types("regression") == ProblemTypes.REGRESSION
>>> assert handle_problem_types("TIME SERIES BINARY") == ProblemTypes.TIME_SERIES_
↪BINARY
>>> assert handle_problem_types("Multiclass") == ProblemTypes.MULTICLASS
```

evalml.problem_types.**is_binary**(*problem_type*)

Determines if the provided problem_type is a binary classification problem type.

**Parameters** **problem_type** (*str or* ProblemTypes) – type of supervised learning problem. See evalml.problem_types.ProblemType.all_problem_types for a full list.

**Returns** Whether or not the provided problem_type is a binary classification problem type.

**Return type** bool

### Examples

```
>>> assert is_binary("Binary")
>>> assert is_binary(ProblemTypes.BINARY)
>>> assert is_binary(ProblemTypes.TIME_SERIES_BINARY)
```

evalml.problem_types.**is_classification**(*problem_type*)

Determines if the provided problem_type is a classification problem type.

**Parameters** **problem_type** (*str or* ProblemTypes) – type of supervised learning problem. See evalml.problem_types.ProblemType.all_problem_types for a full list.

**Returns** Whether or not the provided problem_type is a classification problem type.

**Return type** bool

### Examples

```
>>> assert is_classification("Multiclass")
>>> assert is_classification(ProblemTypes.TIME_SERIES_BINARY)
>>> assert not is_classification(ProblemTypes.REGRESSION)
```

evalml.problem_types.**is_multiclass**(*problem_type*)

Determines if the provided problem_type is a multiclass classification problem type.

**Parameters** **problem_type** (*str or* ProblemTypes) – type of supervised learning problem. See evalml.problem_types.ProblemType.all_problem_types for a full list.

**Returns** Whether or not the provided problem_type is a multiclass classification problem type.

**Return type** bool

### Examples

```
>>> assert is_multiclass("Multiclass")
>>> assert is_multiclass(ProblemTypes.MULTICLASS)
>>> assert is_multiclass(ProblemTypes.TIME_SERIES_MULTICLASS)
```

evalml.problem_types.**is_regression**(*problem_type*)

Determines if the provided problem_type is a regression problem type.

**Parameters** **problem_type** (*str or* ProblemTypes) – type of supervised learning problem. See evalml.problem_types.ProblemType.all_problem_types for a full list.

**Returns** Whether or not the provided problem_type is a regression problem type.

**Return type** bool

### Examples

```
>>> assert is_regression("Regression")
>>> assert is_regression(ProblemTypes.REGRESSION)
>>> assert is_regression(ProblemTypes.TIME_SERIES_REGRESSION)
```

evalml.problem_types.**is_time_series**(*problem_type*)

Determines if the provided problem_type is a time series problem type.

**Parameters** **problem_type** (*str or* ProblemTypes) – type of supervised learning problem. See evalml.problem_types.ProblemType.all_problem_types for a full list.

**Returns** Whether or not the provided problem_type is a time series problem type.

**Return type** bool

### Examples

```
>>> assert is_time_series("time series regression")
>>> assert is_time_series(ProblemTypes.TIME_SERIES_BINARY)
>>> assert not is_time_series(ProblemTypes.REGRESSION)
```

**class** evalml.problem_types.**ProblemTypes**

Enum defining the supported types of machine learning problems.

**Attributes**

| | |
|---|---|
| **BINARY** | Binary classification problem. |
| **MULTI-CLASS** | Multiclass classification problem. |
| **REGRES-SION** | Regression problem. |
| **TIME_SERIES_BINARY** | Time series binary classification problem. |
| **TIME_SERIES_MULTICLASS** | Time series multiclass classification problem. |
| **TIME_SERIES_REGRESSION** | Time series regression problem. |

**Methods**

| | |
|---|---|
| *all_problem_types* | Get a list of all defined problem types. |
| *name* | The name of the Enum member. |
| *value* | The value of the Enum member. |

**all_problem_types**(*cls*)

Get a list of all defined problem types.

**Returns** List of all defined problem types.

**Return type** list(*ProblemTypes*)

**name**(*self*)

The name of the Enum member.

**value**(*self*)

The value of the Enum member.

## Tuners

EvalML tuner classes.

## Submodules

## grid_search_tuner

Grid Search Optimizer, which generates all of the possible points to search for using a grid.

## Module Contents

## Classes Summary

| | |
|---|---|
| *GridSearchTuner* | Grid Search Optimizer, which generates all of the possible points to search for using a grid. |

## Contents

**class** evalml.tuners.grid_search_tuner.**GridSearchTuner**(*pipeline_hyperparameter_ranges*, *n_points=10*, *random_seed=0*)

Grid Search Optimizer, which generates all of the possible points to search for using a grid.

**Parameters**

- **pipeline_hyperparameter_ranges** (*dict*) – a set of hyperparameter ranges corresponding to a pipeline's parameters
- **n_points** (*int*) – The number of points to sample from along each dimension defined in the space argument. Defaults to 10.
- **random_seed** (*int*) – Seed for random number generator. Unused in this class, defaults to 0.

### Examples

```
>>> tuner = GridSearchTuner({'My Component': {'param a': [0.0, 10.0], 'param b': ['a
↪', 'b', 'c']}}, n_points=5)
>>> proposal = tuner.propose()
...
>>> assert proposal.keys() == {'My Component'}
>>> assert proposal['My Component'] == {'param a': 0.0, 'param b': 'a'}
```

Determines points using a grid search approach.

```
>>> for each in range(5):
...     print(tuner.propose())
{'My Component': {'param a': 0.0, 'param b': 'b'}}
{'My Component': {'param a': 0.0, 'param b': 'c'}}
{'My Component': {'param a': 10.0, 'param b': 'a'}}
{'My Component': {'param a': 10.0, 'param b': 'b'}}
{'My Component': {'param a': 10.0, 'param b': 'c'}}
```

**Methods**

| | |
|---|---|
| *add* | Not applicable to grid search tuner as generated parameters are not dependent on scores of previous parameters. |
| *get_starting_parameters* | Gets the starting parameters given the pipeline hyperparameter range. |
| *is_search_space_exhausted* | Checks if it is possible to generate a set of valid parameters. Stores generated parameters in `self.curr_params` to be returned by `propose()`. |
| *propose* | Returns parameters from _grid_points iterations. |

**add**(*self*, *pipeline_parameters*, *score*)

Not applicable to grid search tuner as generated parameters are not dependent on scores of previous parameters.

**Parameters**

- **pipeline_parameters** (*dict*) – a dict of the parameters used to evaluate a pipeline

- **score** (*float*) – the score obtained by evaluating the pipeline with the provided parameters

**get_starting_parameters**(*self*, *hyperparameter_ranges*, *random_seed=0*)

Gets the starting parameters given the pipeline hyperparameter range.

**Parameters**

- **hyperparameter_ranges** (*dict*) – The custom hyperparameter ranges passed in during search. Used to determine the starting parameters.
- **random_seed** (*int*) – The random seed to use. Defaults to 0.

**Returns** The starting parameters, randomly chosen, to initialize a pipeline with.

**Return type** dict

**is_search_space_exhausted**(*self*)

Checks if it is possible to generate a set of valid parameters. Stores generated parameters in `self.curr_params` to be returned by `propose()`.

**Returns** If no more valid parameters exists in the search space, return False.

**Return type** bool

**Raises** **NoParamsException** – If a search space is exhausted, then this exception is thrown.

**propose**(*self*)

Returns parameters from _grid_points iterations.

If all possible combinations of parameters have been scored, then `NoParamsException` is raised.

**Returns** proposed pipeline parameters

**Return type** dict

### random_search_tuner

Random Search Optimizer.

### Module Contents

### Classes Summary

| | |
|---|---|
| *RandomSearchTuner* | Random Search Optimizer. |

### Contents

**class** evalml.tuners.random_search_tuner.**RandomSearchTuner**(*pipeline_hyperparameter_ranges*, *with_replacement=False*, *replacement_max_attempts=10*, *random_seed=0*)

Random Search Optimizer.

**Parameters**

- **pipeline_hyperparameter_ranges** (*dict*) – a set of hyperparameter ranges corresponding to a pipeline's parameters

- **with_replacement** (*bool*) – If false, only unique hyperparameters will be shown
- **replacement_max_attempts** (*int*) – The maximum number of tries to get a unique set of random parameters. Only used if tuner is initalized with with_replacement=True
- **random_seed** (*int*) – Seed for random number generator. Defaults to 0.

### Example

```
>>> tuner = RandomSearchTuner({'My Component': {'param a': [0.0, 10.0], 'param b': [
↪'a', 'b', 'c']}}, random_seed=42)
>>> proposal = tuner.propose()
...
>>> assert proposal.keys() == {'My Component'}
>>> assert proposal['My Component'] == {'param a': 3.7454011884736254, 'param b': 'c
↪'}
```

Determines points using a random search approach.

```
>>> for each in range(7):
...     print(tuner.propose())
{'My Component': {'param a': 7.3199394181140525, 'param b': 'b'}}
{'My Component': {'param a': 1.5601864044243654, 'param b': 'a'}}
{'My Component': {'param a': 0.5808361216819947, 'param b': 'c'}}
{'My Component': {'param a': 6.011150117432089, 'param b': 'c'}}
{'My Component': {'param a': 0.2058449429580245, 'param b': 'c'}}
{'My Component': {'param a': 8.32442640800422, 'param b': 'a'}}
{'My Component': {'param a': 1.8182496720710064, 'param b': 'a'}}
```

**Methods**

| | |
|---|---|
| *add* | Not applicable to random search tuner as generated parameters are not dependent on scores of previous parameters. |
| *get_starting_parameters* | Gets the starting parameters given the pipeline hyperparameter range. |
| *is_search_space_exhausted* | Checks if it is possible to generate a set of valid parameters. Stores generated parameters in `self.curr_params` to be returned by `propose()`. |
| *propose* | Generate a unique set of parameters. |

**add**(*self*, *pipeline_parameters*, *score*)

Not applicable to random search tuner as generated parameters are not dependent on scores of previous parameters.

**Parameters**

- **pipeline_parameters** (*dict*) – A dict of the parameters used to evaluate a pipeline
- **score** (*float*) – The score obtained by evaluating the pipeline with the provided parameters

**get_starting_parameters**(*self*, *hyperparameter_ranges*, *random_seed=0*)

Gets the starting parameters given the pipeline hyperparameter range.

**Parameters**

- **hyperparameter_ranges** (*dict*) – The custom hyperparameter ranges passed in during search. Used to determine the starting parameters.
- **random_seed** (*int*) – The random seed to use. Defaults to 0.

**Returns** The starting parameters, randomly chosen, to initialize a pipeline with.

**Return type** dict

**is_search_space_exhausted**(*self*)

Checks if it is possible to generate a set of valid parameters. Stores generated parameters in `self.curr_params` to be returned by `propose()`.

**Returns** If no more valid parameters exists in the search space, return False.

**Return type** bool

**Raises** **NoParamsException** – If a search space is exhausted, then this exception is thrown.

**propose**(*self*)

Generate a unique set of parameters.

If tuner was initialized with `with_replacement=True` and the tuner is unable to generate a unique set of parameters after `replacement_max_attempts` tries, then `NoParamsException` is raised.

**Returns** Proposed pipeline parameters

**Return type** dict

## skopt_tuner

Bayesian Optimizer.

### Module Contents

### Classes Summary

| | |
|---|---|
| *SKOptTuner* | Bayesian Optimizer. |

### Attributes Summary

| |
|---|
| *logger* |

### Contents

`evalml.tuners.skopt_tuner.logger`

**class** `evalml.tuners.skopt_tuner.SKOptTuner`(*pipeline_hyperparameter_ranges*, *random_seed=0*)

Bayesian Optimizer.

**Parameters**

- **pipeline_hyperparameter_ranges** (*dict*) – A set of hyperparameter ranges corresponding to a pipeline's parameters.
- **random_seed** (*int*) – The seed for the random number generator. Defaults to 0.

### Examples

```
>>> tuner = SKOptTuner({'My Component': {'param a': [0.0, 10.0], 'param b': ['a', 'b
↪', 'c']}})
>>> proposal = tuner.propose()
...
>>> assert proposal.keys() == {'My Component'}
>>> assert proposal['My Component'] == {'param a': 5.928446182250184, 'param b': 'c
↪'}
```

Determines points using a Bayesian Optimizer approach.

```
>>> for each in range(7):
...     print(tuner.propose())
{'My Component': {'param a': 8.57945617622757, 'param b': 'c'}}
{'My Component': {'param a': 6.235636967859724, 'param b': 'b'}}
{'My Component': {'param a': 2.9753460654447235, 'param b': 'a'}}
{'My Component': {'param a': 2.7265629458011325, 'param b': 'b'}}
{'My Component': {'param a': 8.121687287754932, 'param b': 'b'}}
{'My Component': {'param a': 3.927847961008298, 'param b': 'c'}}
{'My Component': {'param a': 3.3739616041726843, 'param b': 'b'}}
```

**Methods**

| | |
|---|---|
| *add* | Add score to sample. |
| *get_starting_parameters* | Gets the starting parameters given the pipeline hyperparameter range. |
| *is_search_space_exhausted* | Optional. If possible search space for tuner is finite, this method indicates whether or not all possible parameters have been scored. |
| *propose* | Returns a suggested set of parameters to train and score a pipeline with, based off the search space dimensions and prior samples. |

**add**(*self*, *pipeline_parameters*, *score*)

Add score to sample.

**Parameters**

- **pipeline_parameters** (*dict*) – A dict of the parameters used to evaluate a pipeline

- **score** (*float*) – The score obtained by evaluating the pipeline with the provided parameters

**Returns** None

**Raises**

- **Exception** – If skopt tuner errors.
- **ParameterError** – If skopt receives invalid parameters.

**get_starting_parameters**(*self*, *hyperparameter_ranges*, *random_seed=0*)

Gets the starting parameters given the pipeline hyperparameter range.

**Parameters**

- **hyperparameter_ranges** (*dict*) – The custom hyperparameter ranges passed in during search. Used to determine the starting parameters.
- **random_seed** (*int*) – The random seed to use. Defaults to 0.

**Returns** The starting parameters, randomly chosen, to initialize a pipeline with.

**Return type** dict

**is_search_space_exhausted**(*self*)

Optional. If possible search space for tuner is finite, this method indicates whether or not all possible parameters have been scored.

**Returns** Returns true if all possible parameters in a search space has been scored.

**Return type** bool

**propose**(*self*)

Returns a suggested set of parameters to train and score a pipeline with, based off the search space dimensions and prior samples.

**Returns** Proposed pipeline parameters.

**Return type** dict

### tuner

Base Tuner class.

### Module Contents

### Classes Summary

| | |
|---|---|
| *Tuner* | Base Tuner class. |

## Contents

**class** `evalml.tuners.tuner.Tuner`(*pipeline_hyperparameter_ranges*, *random_seed=0*)

Base Tuner class.

Tuners implement different strategies for sampling from a search space. They're used in EvalML to search the space of pipeline hyperparameters.

**Parameters**

- **pipeline_hyperparameter_ranges** (*dict*) – a set of hyperparameter ranges corresponding to a pipeline's parameters.
- **random_seed** (*int*) – The random state. Defaults to 0.

**Methods**

| | |
|---|---|
| *add* | Register a set of hyperparameters with the score obtained from training a pipeline with those hyperparameters. |
| *get_starting_parameters* | Gets the starting parameters given the pipeline hyperparameter range. |
| *is_search_space_exhausted* | Optional. If possible search space for tuner is finite, this method indicates whether or not all possible parameters have been scored. |
| *propose* | Returns a suggested set of parameters to train and score a pipeline with, based off the search space dimensions and prior samples. |

**abstract add**(*self*, *pipeline_parameters*, *score*)

Register a set of hyperparameters with the score obtained from training a pipeline with those hyperparameters.

**Parameters**

- **pipeline_parameters** (*dict*) – a dict of the parameters used to evaluate a pipeline
- **score** (*float*) – the score obtained by evaluating the pipeline with the provided parameters

**Returns** None

**get_starting_parameters**(*self*, *hyperparameter_ranges*, *random_seed=0*)

Gets the starting parameters given the pipeline hyperparameter range.

**Parameters**

- **hyperparameter_ranges** (*dict*) – The custom hyperparameter ranges passed in during search. Used to determine the starting parameters.
- **random_seed** (*int*) – The random seed to use. Defaults to 0.

**Returns** The starting parameters, randomly chosen, to initialize a pipeline with.

**Return type** dict

**is_search_space_exhausted**(*self*)

Optional. If possible search space for tuner is finite, this method indicates whether or not all possible parameters have been scored.

**Returns** Returns true if all possible parameters in a search space has been scored.

**Return type** bool

**abstract propose**(*self*)

Returns a suggested set of parameters to train and score a pipeline with, based off the search space dimensions and prior samples.

**Returns** Proposed pipeline parameters

**Return type** dict

### tuner_exceptions

Exception thrown by tuner classes.

### Module Contents

### Contents

**exception** evalml.tuners.tuner_exceptions.**NoParamsException**

Raised when a tuner exhausts its search space and runs out of parameters to propose.

**exception** evalml.tuners.tuner_exceptions.**ParameterError**

Raised when a tuner encounters an error with the parameters being used with it.

### Package Contents

### Classes Summary

| | |
|---|---|
| *GridSearchTuner* | Grid Search Optimizer, which generates all of the possible points to search for using a grid. |
| *RandomSearchTuner* | Random Search Optimizer. |
| *SKOptTuner* | Bayesian Optimizer. |
| *Tuner* | Base Tuner class. |

### Exceptions Summary

### Contents

**class** evalml.tuners.**GridSearchTuner**(*pipeline_hyperparameter_ranges*, *n_points=10*, *random_seed=0*)

Grid Search Optimizer, which generates all of the possible points to search for using a grid.

**Parameters**

- **pipeline_hyperparameter_ranges** (*dict*) – a set of hyperparameter ranges corresponding to a pipeline's parameters
- **n_points** (*int*) – The number of points to sample from along each dimension defined in the space argument. Defaults to 10.

- **random_seed** (*int*) – Seed for random number generator. Unused in this class, defaults to 0.

### Examples

```
>>> tuner = GridSearchTuner({'My Component': {'param a': [0.0, 10.0], 'param b': ['a
↪', 'b', 'c']}}, n_points=5)
>>> proposal = tuner.propose()
...
>>> assert proposal.keys() == {'My Component'}
>>> assert proposal['My Component'] == {'param a': 0.0, 'param b': 'a'}
```

Determines points using a grid search approach.

```
>>> for each in range(5):
...     print(tuner.propose())
{'My Component': {'param a': 0.0, 'param b': 'b'}}
{'My Component': {'param a': 0.0, 'param b': 'c'}}
{'My Component': {'param a': 10.0, 'param b': 'a'}}
{'My Component': {'param a': 10.0, 'param b': 'b'}}
{'My Component': {'param a': 10.0, 'param b': 'c'}}
```

**Methods**

| | |
|---|---|
| *add* | Not applicable to grid search tuner as generated parameters are not dependent on scores of previous parameters. |
| *get_starting_parameters* | Gets the starting parameters given the pipeline hyperparameter range. |
| *is_search_space_exhausted* | Checks if it is possible to generate a set of valid parameters. Stores generated parameters in `self.curr_params` to be returned by `propose()`. |
| *propose* | Returns parameters from _grid_points iterations. |

**add**(*self*, *pipeline_parameters*, *score*)

Not applicable to grid search tuner as generated parameters are not dependent on scores of previous parameters.

**Parameters**

- **pipeline_parameters** (*dict*) – a dict of the parameters used to evaluate a pipeline
- **score** (*float*) – the score obtained by evaluating the pipeline with the provided parameters

**get_starting_parameters**(*self*, *hyperparameter_ranges*, *random_seed=0*)

Gets the starting parameters given the pipeline hyperparameter range.

**Parameters**

- **hyperparameter_ranges** (*dict*) – The custom hyperparameter ranges passed in during search. Used to determine the starting parameters.
- **random_seed** (*int*) – The random seed to use. Defaults to 0.

**Returns** The starting parameters, randomly chosen, to initialize a pipeline with.

**Return type** dict

**is_search_space_exhausted**(*self*)

Checks if it is possible to generate a set of valid parameters. Stores generated parameters in `self.curr_params` to be returned by `propose()`.

**Returns** If no more valid parameters exists in the search space, return False.

**Return type** bool

**Raises** *NoParamsException* – If a search space is exhausted, then this exception is thrown.

**propose**(*self*)

Returns parameters from _grid_points iterations.

If all possible combinations of parameters have been scored, then `NoParamsException` is raised.

**Returns** proposed pipeline parameters

**Return type** dict

**exception** evalml.tuners.**NoParamsException**

Raised when a tuner exhausts its search space and runs out of parameters to propose.

**exception** evalml.tuners.**ParameterError**

Raised when a tuner encounters an error with the parameters being used with it.

**class** evalml.tuners.**RandomSearchTuner**(*pipeline_hyperparameter_ranges*, *with_replacement=False*, *replacement_max_attempts=10*, *random_seed=0*)

Random Search Optimizer.

**Parameters**

- **pipeline_hyperparameter_ranges** (*dict*) – a set of hyperparameter ranges corresponding to a pipeline's parameters
- **with_replacement** (*bool*) – If false, only unique hyperparameters will be shown
- **replacement_max_attempts** (*int*) – The maximum number of tries to get a unique set of random parameters. Only used if tuner is initalized with with_replacement=True
- **random_seed** (*int*) – Seed for random number generator. Defaults to 0.

### Example

```
>>> tuner = RandomSearchTuner({'My Component': {'param a': [0.0, 10.0], 'param b': [
↪'a', 'b', 'c']}}, random_seed=42)
>>> proposal = tuner.propose()
...
>>> assert proposal.keys() == {'My Component'}
>>> assert proposal['My Component'] == {'param a': 3.7454011884736254, 'param b': 'c
↪'}
```

Determines points using a random search approach.

```
>>> for each in range(7):
...     print(tuner.propose())
{'My Component': {'param a': 7.3199394181140525, 'param b': 'b'}}
{'My Component': {'param a': 1.5601864044243654, 'param b': 'a'}}
{'My Component': {'param a': 0.5808361216819947, 'param b': 'c'}}
{'My Component': {'param a': 6.011150117432089, 'param b': 'c'}}
```

(continues on next page)

(continued from previous page)

```
{'My Component': {'param a': 0.2058449429580245, 'param b': 'c'}}
{'My Component': {'param a': 8.32442640800422, 'param b': 'a'}}
{'My Component': {'param a': 1.8182496720710064, 'param b': 'a'}}
```

**Methods**

| | |
|---|---|
| *add* | Not applicable to random search tuner as generated parameters are not dependent on scores of previous parameters. |
| *get_starting_parameters* | Gets the starting parameters given the pipeline hyperparameter range. |
| *is_search_space_exhausted* | Checks if it is possible to generate a set of valid parameters. Stores generated parameters in `self.curr_params` to be returned by `propose()`. |
| *propose* | Generate a unique set of parameters. |

**add**(*self*, *pipeline_parameters*, *score*)

Not applicable to random search tuner as generated parameters are not dependent on scores of previous parameters.

**Parameters**

- **pipeline_parameters** (*dict*) – A dict of the parameters used to evaluate a pipeline
- **score** (*float*) – The score obtained by evaluating the pipeline with the provided parameters

**get_starting_parameters**(*self*, *hyperparameter_ranges*, *random_seed=0*)

Gets the starting parameters given the pipeline hyperparameter range.

**Parameters**

- **hyperparameter_ranges** (*dict*) – The custom hyperparameter ranges passed in during search. Used to determine the starting parameters.
- **random_seed** (*int*) – The random seed to use. Defaults to 0.

**Returns** The starting parameters, randomly chosen, to initialize a pipeline with.

**Return type** dict

**is_search_space_exhausted**(*self*)

Checks if it is possible to generate a set of valid parameters. Stores generated parameters in `self.curr_params` to be returned by `propose()`.

**Returns** If no more valid parameters exists in the search space, return False.

**Return type** bool

**Raises** *NoParamsException* – If a search space is exhausted, then this exception is thrown.

**propose**(*self*)

Generate a unique set of parameters.

If tuner was initialized with `with_replacement=True` and the tuner is unable to generate a unique set of parameters after `replacement_max_attempts` tries, then `NoParamsException` is raised.

**Returns** Proposed pipeline parameters

**Return type** dict

**class** evalml.tuners.**SKOptTuner**(*pipeline_hyperparameter_ranges*, *random_seed=0*)

Bayesian Optimizer.

**Parameters**

- **pipeline_hyperparameter_ranges** (*dict*) – A set of hyperparameter ranges corresponding to a pipeline's parameters.
- **random_seed** (*int*) – The seed for the random number generator. Defaults to 0.

### Examples

```
>>> tuner = SKOptTuner({'My Component': {'param a': [0.0, 10.0], 'param b': ['a', 'b
↪', 'c']}})
>>> proposal = tuner.propose()
...
>>> assert proposal.keys() == {'My Component'}
>>> assert proposal['My Component'] == {'param a': 5.928446182250184, 'param b': 'c
↪'}
```

Determines points using a Bayesian Optimizer approach.

```
>>> for each in range(7):
...     print(tuner.propose())
{'My Component': {'param a': 8.57945617622757, 'param b': 'c'}}
{'My Component': {'param a': 6.235636967859724, 'param b': 'b'}}
{'My Component': {'param a': 2.9753460654447235, 'param b': 'a'}}
{'My Component': {'param a': 2.7265629458011325, 'param b': 'b'}}
{'My Component': {'param a': 8.121687287754932, 'param b': 'b'}}
{'My Component': {'param a': 3.927847961008298, 'param b': 'c'}}
{'My Component': {'param a': 3.3739616041726843, 'param b': 'b'}}
```

**Methods**

| | |
|---|---|
| *add* | Add score to sample. |
| *get_starting_parameters* | Gets the starting parameters given the pipeline hyperparameter range. |
| *is_search_space_exhausted* | Optional. If possible search space for tuner is finite, this method indicates whether or not all possible parameters have been scored. |
| *propose* | Returns a suggested set of parameters to train and score a pipeline with, based off the search space dimensions and prior samples. |

**add**(*self*, *pipeline_parameters*, *score*)

Add score to sample.

**Parameters**

- **pipeline_parameters** (*dict*) – A dict of the parameters used to evaluate a pipeline
- **score** (*float*) – The score obtained by evaluating the pipeline with the provided parameters

**Returns** None

**Raises**

- **Exception** – If skopt tuner errors.
- *ParameterError* – If skopt receives invalid parameters.

**get_starting_parameters**(*self*, *hyperparameter_ranges*, *random_seed=0*)

Gets the starting parameters given the pipeline hyperparameter range.

**Parameters**

- **hyperparameter_ranges** (*dict*) – The custom hyperparameter ranges passed in during search. Used to determine the starting parameters.
- **random_seed** (*int*) – The random seed to use. Defaults to 0.

**Returns** The starting parameters, randomly chosen, to initialize a pipeline with.

**Return type** dict

**is_search_space_exhausted**(*self*)

Optional. If possible search space for tuner is finite, this method indicates whether or not all possible parameters have been scored.

**Returns** Returns true if all possible parameters in a search space has been scored.

**Return type** bool

**propose**(*self*)

Returns a suggested set of parameters to train and score a pipeline with, based off the search space dimensions and prior samples.

**Returns** Proposed pipeline parameters.

**Return type** dict

**class** evalml.tuners.**Tuner**(*pipeline_hyperparameter_ranges*, *random_seed=0*)

Base Tuner class.

Tuners implement different strategies for sampling from a search space. They're used in EvalML to search the space of pipeline hyperparameters.

**Parameters**

- **pipeline_hyperparameter_ranges** (*dict*) – a set of hyperparameter ranges corresponding to a pipeline's parameters.
- **random_seed** (*int*) – The random state. Defaults to 0.

**Methods**

| | |
|---|---|
| *add* | Register a set of hyperparameters with the score obtained from training a pipeline with those hyperparameters. |
| *get_starting_parameters* | Gets the starting parameters given the pipeline hyperparameter range. |
| *is_search_space_exhausted* | Optional. If possible search space for tuner is finite, this method indicates whether or not all possible parameters have been scored. |
| *propose* | Returns a suggested set of parameters to train and score a pipeline with, based off the search space dimensions and prior samples. |

**abstract add**(*self*, *pipeline_parameters*, *score*)

Register a set of hyperparameters with the score obtained from training a pipeline with those hyperparameters.

**Parameters**

- **pipeline_parameters** (*dict*) – a dict of the parameters used to evaluate a pipeline
- **score** (*float*) – the score obtained by evaluating the pipeline with the provided parameters

**Returns** None

**get_starting_parameters**(*self*, *hyperparameter_ranges*, *random_seed=0*)

Gets the starting parameters given the pipeline hyperparameter range.

**Parameters**

- **hyperparameter_ranges** (*dict*) – The custom hyperparameter ranges passed in during search. Used to determine the starting parameters.
- **random_seed** (*int*) – The random seed to use. Defaults to 0.

**Returns** The starting parameters, randomly chosen, to initialize a pipeline with.

**Return type** dict

**is_search_space_exhausted**(*self*)

Optional. If possible search space for tuner is finite, this method indicates whether or not all possible parameters have been scored.

**Returns** Returns true if all possible parameters in a search space has been scored.

**Return type** bool

**abstract propose**(*self*)

Returns a suggested set of parameters to train and score a pipeline with, based off the search space dimensions and prior samples.

**Returns** Proposed pipeline parameters

**Return type** dict

## Utils

Utility methods.

## Submodules

## base_meta

Metaclass that overrides creating a new component or pipeline by wrapping methods with validators and setters.

### Module Contents

### Classes Summary

| | |
|---|---|
| *BaseMeta* | Metaclass that overrides creating a new component or pipeline by wrapping methods with validators and setters. |

### Contents

**class** evalml.utils.base_meta.**BaseMeta**

Metaclass that overrides creating a new component or pipeline by wrapping methods with validators and setters.

**Attributes**

| | |
|---|---|
| **FIT_METHODS** | ['fit', 'fit_transform'] |
| **METHODS_TO_CHECK** | ['predict', 'predict_proba', 'transform', 'inverse_transform', 'get_trend_dataframe'] |
| **PROPERTIES_TO_CHECK** | ['feature_importance'] |

**Methods**

| | |
|---|---|
| *register* | Register a virtual subclass of an ABC. |
| *set_fit* | Wrapper for the fit method. |

**register**(*cls*, *subclass*)

Register a virtual subclass of an ABC.

Returns the subclass, to allow usage as a class decorator.

**classmethod set_fit**(*cls*, *method*)

Wrapper for the fit method.

## cli_utils

CLI functions.

### Module Contents

### Functions

| | |
|---|---|
| *get_evalml_black_config* | Gets configuration for black. |
| *get_evalml_pip_requirements* | Gets pip requirements for evalml (with pip packages converted to conda names) |
| *get_evalml_requirements_file* | Gets pip requirements for evalml as a requirements file |
| *get_evalml_root* | Gets location where evalml is installed. |
| *get_installed_packages* | Get dictionary mapping installed package names to their versions. |
| *get_sys_info* | Returns system information. |
| *print_deps* | Prints the version number of each dependency. |
| *print_info* | Prints information about the system, evalml, and dependencies of evalml. |
| *print_sys_info* | Prints system information. |
| *standardize_format* | Standardizes the format of the given packages. |

### Attributes Summary

| |
|---|
| *CONDA_TO_PIP_NAME* |

### Contents

evalml.utils.cli_utils.**CONDA_TO_PIP_NAME**

evalml.utils.cli_utils.**get_evalml_black_config**(*evalml_path*)

Gets configuration for black.

**Parameters** **evalml_path** – Path to evalml root.

**Returns** Dictionary of black configuration.

evalml.utils.cli_utils.**get_evalml_pip_requirements**(*evalml_path*, *ignore_packages=None*, *convert_to_conda=True*)

Gets pip requirements for evalml (with pip packages converted to conda names)

**Parameters**

- **evalml_path** – Path to evalml root.
- **ignore_packages** – List of packages to ignore. Defaults to None.

**Returns** List of pip requirements for evalml.

evalml.utils.cli_utils.**get_evalml_requirements_file**(*evalml_path*, *requirements_file_path*)

Gets pip requirements for evalml as a requirements file

**Parameters**

- **evalml_path** – Path to evalml root.
- **requirements_file_path** – Path to requirements file.

**Returns** Pip requirements for evalml in a singular string.

`evalml.utils.cli_utils.get_evalml_root()`

Gets location where evalml is installed.

**Returns** Location where evalml is installed.

`evalml.utils.cli_utils.get_installed_packages()`

Get dictionary mapping installed package names to their versions.

**Returns** Dictionary mapping installed package names to their versions.

`evalml.utils.cli_utils.get_sys_info()`

Returns system information.

**Returns** List of tuples about system stats.

`evalml.utils.cli_utils.print_deps()`

Prints the version number of each dependency.

`evalml.utils.cli_utils.print_info()`

Prints information about the system, evalml, and dependencies of evalml.

`evalml.utils.cli_utils.print_sys_info()`

Prints system information.

`evalml.utils.cli_utils.standardize_format(packages, ignore_packages=None, convert_to_conda=True)`

Standardizes the format of the given packages.

**Parameters**

- **packages** – Requirements package generator object.
- **ignore_packages** – List of packages to ignore. Defaults to None.

**Returns** List of packages with standardized format.

## gen_utils

General utility methods.

### Module Contents

### Classes Summary

| | |
|---|---|
| *classproperty* | Allows function to be accessed as a class level property. |

### Functions

| | |
|---|---|
| *are_datasets_separated_by_gap_time_index* | Determine if the train and test datasets are separated by gap number of units using the time_index. |
| *are_ts_parameters_valid_for_split* | Validates the time series parameters in problem_configuration are compatible with split sizes. |
| *contains_all_ts_parameters* | Validates that the problem configuration contains all required keys. |
| *convert_to_seconds* | Converts a string describing a length of time to its length in seconds. |
| *deprecate_arg* | Helper to raise warnings when a deprecated arg is used. |
| *drop_rows_with_nans* | Drop rows that have any NaNs in all dataframes or series. |
| *get_importable_subclasses* | Get importable subclasses of a base class. Used to list all of our estimators, transformers, components and pipelines dynamically. |
| *get_random_seed* | Given a numpy.random.RandomState object, generate an int representing a seed value for another random number generator. Or, if given an int, return that int. |
| *get_random_state* | Generates a numpy.random.RandomState instance using seed. |
| *get_time_index* | Determines the column in the given data that should be used as the time index. |
| *import_or_raise* | Attempts to import the requested library by name. If the import fails, raises an ImportError or warning. |
| *is_all_numeric* | Checks if the given DataFrame contains only numeric values. |
| *jupyter_check* | Get whether or not the code is being run in a Ipython environment (such as Jupyter Notebook or Jupyter Lab). |
| *pad_with_nans* | Pad the beginning num_to_pad rows with nans. |
| *safe_repr* | Convert the given value into a string that can safely be used for repr. |
| *save_plot* | Saves fig to filepath if specified, or to a default location if not. |
| *validate_holdout_datasets* | Validate the holdout datasets match our expectations. |

### Attributes Summary

| |
|---|
| *logger* |
| *SEED_BOUNDS* |

## Contents

evalml.utils.gen_utils.**are_datasets_separated_by_gap_time_index**(*train*, *test*, *pipeline_params*, *freq=None*)

Determine if the train and test datasets are separated by gap number of units using the time_index.

This will be true when users are predicting on unseen data but not during cross validation since the target is known.

**Parameters**

- **train** (*pd.DataFrame*) – Training data.
- **test** (*pd.DataFrame*) – Data of shape [n_samples, n_features].
- **pipeline_params** (*dict*) – Dictionary of time series parameters.
- **freq** (*str*) – Frequency of time index.

**Returns** True if the difference in time units is equal to gap + 1.

**Return type** bool

evalml.utils.gen_utils.**are_ts_parameters_valid_for_split**(*gap*, *max_delay*, *forecast_horizon*, *n_obs*, *n_splits*)

Validates the time series parameters in problem_configuration are compatible with split sizes.

**Parameters**

- **gap** (*int*) – gap value.
- **max_delay** (*int*) – max_delay value.
- **forecast_horizon** (*int*) – forecast_horizon value.
- **n_obs** (*int*) – Number of observations in the dataset.
- **n_splits** (*int*) – Number of cross validation splits.

**Returns**

**TsParameterValidationResult - named tuple with four fields** is_valid (bool): True if parameters are valid. msg (str): Contains error message to display. Empty if is_valid. smallest_split_size (int): Smallest split size given n_obs and n_splits. max_window_size (int): Max window size given gap, max_delay, forecast_horizon.

**class** evalml.utils.gen_utils.**classproperty**(*func*)

Allows function to be accessed as a class level property.

Example: .. code-block:

```
class LogisticRegressionBinaryPipeline(PipelineBase):
    component_graph = ['Simple Imputer', 'Logistic Regression Classifier']

    @classproperty
    def summary(cls):
    summary = ""
    for component in cls.component_graph:
        component = handle_component_class(component)
        summary += component.name + " + "
    return summary
```

(continues on next page)

(continued from previous page)

```
assert LogisticRegressionBinaryPipeline.summary == "Simple Imputer + Logistic␣
↪Regression Classifier + "
assert LogisticRegressionBinaryPipeline().summary == "Simple Imputer + Logistic␣
↪Regression Classifier + "
```

evalml.utils.gen_utils.**contains_all_ts_parameters**(*problem_configuration*)

Validates that the problem configuration contains all required keys.

**Parameters** **problem_configuration** (*dict*) – Problem configuration.

**Returns**

**True if the configuration contains all parameters. If False, msg is a non-empty** string with error message.

**Return type** bool, str

evalml.utils.gen_utils.**convert_to_seconds**(*input_str*)

Converts a string describing a length of time to its length in seconds.

**Parameters** **input_str** (*str*) – The string to be parsed and converted to seconds.

**Returns** Returns the library if importing succeeded.

**Raises** **AssertionError** – If an invalid unit is used.

### Examples

```
>>> assert convert_to_seconds("10 hr") == 36000.0
>>> assert convert_to_seconds("30 minutes") == 1800.0
>>> assert convert_to_seconds("2.5 min") == 150.0
```

evalml.utils.gen_utils.**deprecate_arg**(*old_arg*, *new_arg*, *old_value*, *new_value*)

Helper to raise warnings when a deprecated arg is used.

**Parameters**

- **old_arg** (*str*) – Name of old/deprecated argument.
- **new_arg** (*str*) – Name of new argument.
- **old_value** (*Any*) – Value the user passed in for the old argument.
- **new_value** (*Any*) – Value the user passed in for the new argument.

**Returns** old_value if not None, else new_value

evalml.utils.gen_utils.**drop_rows_with_nans**(**pd_data*)

Drop rows that have any NaNs in all dataframes or series.

**Parameters** ***pd_data** – sequence of pd.Series or pd.DataFrame or None

**Returns** list of pd.DataFrame or pd.Series or None

evalml.utils.gen_utils.**get_importable_subclasses**(*base_class*, *used_in_automl=True*)

Get importable subclasses of a base class. Used to list all of our estimators, transformers, components and pipelines dynamically.

**Parameters**

- **base_class** (*abc.ABCMeta*) – Base class to find all of the subclasses for.

- **used_in_automl** – Not all components/pipelines/estimators are used in automl search. If True, only include those subclasses that are used in the search. This would mean excluding classes related to ExtraTrees, ElasticNet, and Baseline estimators.

**Returns** List of subclasses.

`evalml.utils.gen_utils.get_random_seed(random_state, min_bound=SEED_BOUNDS.min_bound, max_bound=SEED_BOUNDS.max_bound)`

Given a numpy.random.RandomState object, generate an int representing a seed value for another random number generator. Or, if given an int, return that int.

To protect against invalid input to a particular library's random number generator, if an int value is provided, and it is outside the bounds "[min_bound, max_bound)", the value will be projected into the range between the min_bound (inclusive) and max_bound (exclusive) using modular arithmetic.

**Parameters**

- **random_state** (*int, numpy.random.RandomState*) – random state
- **min_bound** (*None, int*) – if not default of None, will be min bound when generating seed (inclusive). Must be less than max_bound.
- **max_bound** (*None, int*) – if not default of None, will be max bound when generating seed (exclusive). Must be greater than min_bound.

**Returns** Seed for random number generator

**Return type** int

**Raises** **ValueError** – If boundaries are not valid.

`evalml.utils.gen_utils.get_random_state(seed)`

Generates a numpy.random.RandomState instance using seed.

**Parameters** **seed** (*None, int, np.random.RandomState object*) – seed to use to generate numpy.random.RandomState. Must be between SEED_BOUNDS.min_bound and SEED_BOUNDS.max_bound, inclusive.

**Raises** **ValueError** – If the input seed is not within the acceptable range.

**Returns** A numpy.random.RandomState instance.

`evalml.utils.gen_utils.get_time_index(X: pandas.DataFrame, y: pandas.Series, time_index_name: str)`

Determines the column in the given data that should be used as the time index.

`evalml.utils.gen_utils.import_or_raise(library, error_msg=None, warning=False)`

Attempts to import the requested library by name. If the import fails, raises an ImportError or warning.

**Parameters**

- **library** (*str*) – The name of the library.
- **error_msg** (*str*) – Error message to return if the import fails.
- **warning** (*bool*) – If True, import_or_raise gives a warning instead of ImportError. Defaults to False.

**Returns** Returns the library if importing succeeded.

**Raises**

- **ImportError** – If attempting to import the library fails because the library is not installed.
- **Exception** – If importing the library fails.

evalml.utils.gen_utils.**is_all_numeric**(*df*)

Checks if the given DataFrame contains only numeric values.

**Parameters** **df** (*pd.DataFrame*) – The DataFrame to check data types of.

**Returns** True if all the columns are numeric and are not missing any values, False otherwise.

evalml.utils.gen_utils.**jupyter_check**()

Get whether or not the code is being run in a Ipython environment (such as Jupyter Notebook or Jupyter Lab).

**Returns** True if Ipython, False otherwise.

**Return type** boolean

evalml.utils.gen_utils.**logger**

evalml.utils.gen_utils.**pad_with_nans**(*pd_data*, *num_to_pad*)

Pad the beginning num_to_pad rows with nans.

**Parameters**

- **pd_data** (*pd.DataFrame or pd.Series*) – Data to pad.
- **num_to_pad** (*int*) – Number of nans to pad.

**Returns** pd.DataFrame or pd.Series

evalml.utils.gen_utils.**safe_repr**(*value*)

Convert the given value into a string that can safely be used for repr.

**Parameters** **value** – The item to convert

**Returns** String representation of the value

evalml.utils.gen_utils.**save_plot**(*fig*, *filepath=None*, *format='png'*, *interactive=False*, *return_filepath=False*)

Saves fig to filepath if specified, or to a default location if not.

**Parameters**

- **fig** (*Figure*) – Figure to be saved.
- **filepath** (*str or Path, optional*) – Location to save file. Default is with filename "test_plot".
- **format** (*str*) – Extension for figure to be saved as. Ignored if interactive is True and fig is of type plotly.Figure. Defaults to 'png'.
- **interactive** (*bool, optional*) – If True and fig is of type plotly.Figure, saves the fig as interactive instead of static, and format will be set to 'html'. Defaults to False.
- **return_filepath** (*bool, optional*) – Whether to return the final filepath the image is saved to. Defaults to False.

**Returns** String representing the final filepath the image was saved to if return_filepath is set to True. Defaults to None.

evalml.utils.gen_utils.**SEED_BOUNDS**

evalml.utils.gen_utils.**validate_holdout_datasets**(*X*, *X_train*, *pipeline_params*)

Validate the holdout datasets match our expectations.

This function is run before calling predict in a time series pipeline. It verifies that X (the holdout set) is gap units away from the training set and is less than or equal to the forecast_horizon.

**Parameters**

- **X** (*pd.DataFrame*) – Data of shape [n_samples, n_features].
- **X_train** (*pd.DataFrame*) – Training data.
- **pipeline_params** (*dict*) – Dictionary of time series parameters with gap, forecast_horizon, and time_index being required.

**Returns**

**TSHoldoutValidationResult - named tuple with three fields** is_valid (bool): True if holdout data is valid. error_messages (list): List of error messages to display. Empty if is_valid. error_codes (list): List of error codes to display. Empty if is_valid.

## logger

Logging functions.

### Module Contents

### Functions

| | |
|---|---|
| *get_logger* | Get the logger with the associated name. |
| *log_batch_times* | Used to print out the batch times. |
| *log_subtitle* | Log with a subtitle. |
| *log_title* | Log with a title. |
| *time_elapsed* | How much time has elapsed since the search started. |

### Contents

`evalml.utils.logger.`**`get_logger`**(*name*)

Get the logger with the associated name.

**Parameters** **name** (*str*) – Name of the logger to get.

**Returns** The logger object with the associated name.

`evalml.utils.logger.`**`log_batch_times`**(*logger*, *batch_times*)

Used to print out the batch times.

**Parameters**

- **logger** – the logger.
- **batch_times** – dict with (batch number, {pipeline name, pipeline time}).

`evalml.utils.logger.`**`log_subtitle`**(*logger*, *title*, *underline='='*)

Log with a subtitle.

`evalml.utils.logger.`**`log_title`**(*logger*, *title*)

Log with a title.

`evalml.utils.logger.time_elapsed(start_time)`

How much time has elapsed since the search started.

**Parameters** **start_time** (*int*) – Time when search started.

**Returns** elapsed time formatted as a string [H:]MM:SS

**Return type** str

### nullable_type_utils

#### Module Contents

#### Contents

`evalml.utils.nullable_type_utils.DOWNCAST_TYPE_DICT`

### update_checker

Check if EvalML has updated since the user installed.

#### Module Contents

#### Contents

`evalml.utils.update_checker.method`

### woodwork_utils

Woodwork utility methods.

#### Module Contents

#### Functions

| | |
|---|---|
| *downcast_nullable_types* | Downcasts IntegerNullable, BooleanNullable types to Double, Boolean in order to support certain estimators like ARIMA, CatBoost, and LightGBM. |
| *infer_feature_types* | Create a Woodwork structure from the given list, pandas, or numpy input, with specified types for columns. If a column's type is not specified, it will be inferred by Woodwork. |

### Attributes Summary

| | |
|---|---|
| *numeric_and_boolean_ww* | |

### Contents

`evalml.utils.woodwork_utils.`**`downcast_nullable_types`**(*data*, *ignore_null_cols=True*)

Downcasts IntegerNullable, BooleanNullable types to Double, Boolean in order to support certain estimators like ARIMA, CatBoost, and LightGBM.

**Parameters**

- **data** (*pd.DataFrame, pd.Series*) – Feature data.
- **ignore_null_cols** (*bool*) – Whether to ignore downcasting columns with null values or not. Defaults to True.

**Returns** DataFrame or Series initialized with logical type information where BooleanNullable are cast as Double.

**Return type** data

`evalml.utils.woodwork_utils.`**`infer_feature_types`**(*data*, *feature_types=None*)

Create a Woodwork structure from the given list, pandas, or numpy input, with specified types for columns. If a column's type is not specified, it will be inferred by Woodwork.

**Parameters**

- **data** (*pd.DataFrame, pd.Series*) – Input data to convert to a Woodwork data structure.
- **feature_types** (*string, ww.logical_type obj, dict, optional*) – If data is a 2D structure, feature_types must be a dictionary mapping column names to the type of data represented in the column. If data is a 1D structure, then feature_types must be a Woodwork logical type or a string representing a Woodwork logical type ("Double", "Integer", "Boolean", "Categorical", "Datetime", "NaturalLanguage")

**Returns** A Woodwork data structure where the data type of each column was either specified or inferred.

**Raises** **ValueError** – If there is a mismatch between the dataframe and the woodwork schema.

`evalml.utils.woodwork_utils.`**`numeric_and_boolean_ww`**

### Package Contents

### Classes Summary

| | |
|---|---|
| *classproperty* | Allows function to be accessed as a class level property. |

### Functions

| | |
|---|---|
| *convert_to_seconds* | Converts a string describing a length of time to its length in seconds. |
| *deprecate_arg* | Helper to raise warnings when a deprecated arg is used. |
| *downcast_nullable_types* | Downcasts IntegerNullable, BooleanNullable types to Double, Boolean in order to support certain estimators like ARIMA, CatBoost, and LightGBM. |
| *drop_rows_with_nans* | Drop rows that have any NaNs in all dataframes or series. |
| *get_importable_subclasses* | Get importable subclasses of a base class. Used to list all of our estimators, transformers, components and pipelines dynamically. |
| *get_logger* | Get the logger with the associated name. |
| *get_random_seed* | Given a numpy.random.RandomState object, generate an int representing a seed value for another random number generator. Or, if given an int, return that int. |
| *get_random_state* | Generates a numpy.random.RandomState instance using seed. |
| *get_time_index* | Determines the column in the given data that should be used as the time index. |
| *import_or_raise* | Attempts to import the requested library by name. If the import fails, raises an ImportError or warning. |
| *infer_feature_types* | Create a Woodwork structure from the given list, pandas, or numpy input, with specified types for columns. If a column's type is not specified, it will be inferred by Woodwork. |
| *is_all_numeric* | Checks if the given DataFrame contains only numeric values. |
| *jupyter_check* | Get whether or not the code is being run in a Ipython environment (such as Jupyter Notebook or Jupyter Lab). |
| *log_subtitle* | Log with a subtitle. |
| *log_title* | Log with a title. |
| *pad_with_nans* | Pad the beginning num_to_pad rows with nans. |
| *safe_repr* | Convert the given value into a string that can safely be used for repr. |
| *save_plot* | Saves fig to filepath if specified, or to a default location if not. |

### Attributes Summary

| |
|---|
| *SEED_BOUNDS* |

### Contents

**class** evalml.utils.**classproperty**(*func*)

Allows function to be accessed as a class level property.

Example: .. code-block:

```
class LogisticRegressionBinaryPipeline(PipelineBase):
    component_graph = ['Simple Imputer', 'Logistic Regression Classifier']

    @classproperty
    def summary(cls):
    summary = ""
    for component in cls.component_graph:
        component = handle_component_class(component)
        summary += component.name + " + "
    return summary

assert LogisticRegressionBinaryPipeline.summary == "Simple Imputer + Logistic␣
↪Regression Classifier + "
assert LogisticRegressionBinaryPipeline().summary == "Simple Imputer + Logistic␣
↪Regression Classifier + "
```

evalml.utils.**convert_to_seconds**(*input_str*)

Converts a string describing a length of time to its length in seconds.

**Parameters** **input_str** (*str*) – The string to be parsed and converted to seconds.

**Returns** Returns the library if importing succeeded.

**Raises** **AssertionError** – If an invalid unit is used.

### Examples

```
>>> assert convert_to_seconds("10 hr") == 36000.0
>>> assert convert_to_seconds("30 minutes") == 1800.0
>>> assert convert_to_seconds("2.5 min") == 150.0
```

evalml.utils.**deprecate_arg**(*old_arg*, *new_arg*, *old_value*, *new_value*)

Helper to raise warnings when a deprecated arg is used.

**Parameters**

- **old_arg** (*str*) – Name of old/deprecated argument.
- **new_arg** (*str*) – Name of new argument.
- **old_value** (*Any*) – Value the user passed in for the old argument.
- **new_value** (*Any*) – Value the user passed in for the new argument.

**Returns** old_value if not None, else new_value

evalml.utils.**downcast_nullable_types**(*data*, *ignore_null_cols=True*)

Downcasts IntegerNullable, BooleanNullable types to Double, Boolean in order to support certain estimators like ARIMA, CatBoost, and LightGBM.

**Parameters**

- **data** (*pd.DataFrame, pd.Series*) – Feature data.
- **ignore_null_cols** (*bool*) – Whether to ignore downcasting columns with null values or not. Defaults to True.

**Returns** DataFrame or Series initialized with logical type information where BooleanNullable are cast as Double.

**Return type** data

`evalml.utils.drop_rows_with_nans(*pd_data)`

Drop rows that have any NaNs in all dataframes or series.

**Parameters** ***pd_data** – sequence of pd.Series or pd.DataFrame or None

**Returns** list of pd.DataFrame or pd.Series or None

`evalml.utils.get_importable_subclasses(base_class, used_in_automl=True)`

Get importable subclasses of a base class. Used to list all of our estimators, transformers, components and pipelines dynamically.

**Parameters**

- **base_class** (*abc.ABCMeta*) – Base class to find all of the subclasses for.
- **used_in_automl** – Not all components/pipelines/estimators are used in automl search. If True, only include those subclasses that are used in the search. This would mean excluding classes related to ExtraTrees, ElasticNet, and Baseline estimators.

**Returns** List of subclasses.

`evalml.utils.get_logger(name)`

Get the logger with the associated name.

**Parameters** **name** (*str*) – Name of the logger to get.

**Returns** The logger object with the associated name.

`evalml.utils.get_random_seed(random_state, min_bound=SEED_BOUNDS.min_bound, max_bound=SEED_BOUNDS.max_bound)`

Given a numpy.random.RandomState object, generate an int representing a seed value for another random number generator. Or, if given an int, return that int.

To protect against invalid input to a particular library's random number generator, if an int value is provided, and it is outside the bounds "[min_bound, max_bound)", the value will be projected into the range between the min_bound (inclusive) and max_bound (exclusive) using modular arithmetic.

**Parameters**

- **random_state** (*int, numpy.random.RandomState*) – random state
- **min_bound** (*None, int*) – if not default of None, will be min bound when generating seed (inclusive). Must be less than max_bound.
- **max_bound** (*None, int*) – if not default of None, will be max bound when generating seed (exclusive). Must be greater than min_bound.

**Returns** Seed for random number generator

**Return type** int

**Raises** **ValueError** – If boundaries are not valid.

evalml.utils.**get_random_state**(*seed*)

Generates a numpy.random.RandomState instance using seed.

**Parameters** **seed** (*None, int, np.random.RandomState object*) – seed to use to generate numpy.random.RandomState. Must be between SEED_BOUNDS.min_bound and SEED_BOUNDS.max_bound, inclusive.

**Raises** **ValueError** – If the input seed is not within the acceptable range.

**Returns** A numpy.random.RandomState instance.

evalml.utils.**get_time_index**(*X: pandas.DataFrame*, *y: pandas.Series*, *time_index_name: str*)

Determines the column in the given data that should be used as the time index.

evalml.utils.**import_or_raise**(*library*, *error_msg=None*, *warning=False*)

Attempts to import the requested library by name. If the import fails, raises an ImportError or warning.

**Parameters**

- **library** (*str*) – The name of the library.
- **error_msg** (*str*) – Error message to return if the import fails.
- **warning** (*bool*) – If True, import_or_raise gives a warning instead of ImportError. Defaults to False.

**Returns** Returns the library if importing succeeded.

**Raises**

- **ImportError** – If attempting to import the library fails because the library is not installed.
- **Exception** – If importing the library fails.

evalml.utils.**infer_feature_types**(*data*, *feature_types=None*)

Create a Woodwork structure from the given list, pandas, or numpy input, with specified types for columns. If a column's type is not specified, it will be inferred by Woodwork.

**Parameters**

- **data** (*pd.DataFrame, pd.Series*) – Input data to convert to a Woodwork data structure.
- **feature_types** (*string, ww.logical_type obj, dict, optional*) – If data is a 2D structure, feature_types must be a dictionary mapping column names to the type of data represented in the column. If data is a 1D structure, then feature_types must be a Woodwork logical type or a string representing a Woodwork logical type ("Double", "Integer", "Boolean", "Categorical", "Datetime", "NaturalLanguage")

**Returns** A Woodwork data structure where the data type of each column was either specified or inferred.

**Raises** **ValueError** – If there is a mismatch between the dataframe and the woodwork schema.

evalml.utils.**is_all_numeric**(*df*)

Checks if the given DataFrame contains only numeric values.

**Parameters** **df** (*pd.DataFrame*) – The DataFrame to check data types of.

**Returns** True if all the columns are numeric and are not missing any values, False otherwise.

evalml.utils.**jupyter_check**()

Get whether or not the code is being run in a Ipython environment (such as Jupyter Notebook or Jupyter Lab).

**Returns** True if Ipython, False otherwise.

**Return type** boolean

`evalml.utils.log_subtitle(logger, title, underline='=')`

Log with a subtitle.

`evalml.utils.log_title(logger, title)`

Log with a title.

`evalml.utils.pad_with_nans(pd_data, num_to_pad)`

Pad the beginning num_to_pad rows with nans.

**Parameters**

- **pd_data** (*pd.DataFrame or pd.Series*) – Data to pad.
- **num_to_pad** (*int*) – Number of nans to pad.

**Returns** pd.DataFrame or pd.Series

`evalml.utils.safe_repr(value)`

Convert the given value into a string that can safely be used for repr.

**Parameters** **value** – The item to convert

**Returns** String representation of the value

`evalml.utils.save_plot(fig, filepath=None, format='png', interactive=False, return_filepath=False)`

Saves fig to filepath if specified, or to a default location if not.

**Parameters**

- **fig** (*Figure*) – Figure to be saved.
- **filepath** (*str or Path, optional*) – Location to save file. Default is with filename "test_plot".
- **format** (*str*) – Extension for figure to be saved as. Ignored if interactive is True and fig is of type plotly.Figure. Defaults to 'png'.
- **interactive** (*bool, optional*) – If True and fig is of type plotly.Figure, saves the fig as interactive instead of static, and format will be set to 'html'. Defaults to False.
- **return_filepath** (*bool, optional*) – Whether to return the final filepath the image is saved to. Defaults to False.

**Returns** String representing the final filepath the image was saved to if return_filepath is set to True. Defaults to None.

`evalml.utils.SEED_BOUNDS`

## Package Contents

## Classes Summary

| | |
|---|---|
| *AutoMLSearch* | Automated Pipeline search. |

### Functions

| | |
|---|---|
| *search* | Given data and configuration, run an automl search. |
| *search_iterative* | Given data and configuration, run an automl search. |

### Contents

**class** evalml.**AutoMLSearch**(*X_train=None*, *y_train=None*, *X_holdout=None*, *y_holdout=None*, *problem_type=None*, *objective='auto'*, *max_iterations=None*, *max_time=None*, *patience=None*, *tolerance=None*, *data_splitter=None*, *allowed_component_graphs=None*, *allowed_model_families=None*, *features=None*, *start_iteration_callback=None*, *add_result_callback=None*, *error_callback=None*, *additional_objectives=None*, *alternate_thresholding_objective='F1'*, *random_seed=0*, *n_jobs=- 1*, *tuner_class=None*, *optimize_thresholds=True*, *ensembling=False*, *max_batches=None*, *problem_configuration=None*, *train_best_pipeline=True*, *search_parameters=None*, *sampler_method='auto'*, *sampler_balanced_ratio=0.25*, *allow_long_running_models=False*, *_pipelines_per_batch=5*, *automl_algorithm='default'*, *engine='sequential'*, *verbose=False*, *timing=False*, *exclude_featurizers=None*, *holdout_set_size=0*)

Automated Pipeline search.

**Parameters**

- **X_train** (*pd.DataFrame*) – The input training data of shape [n_samples, n_features]. Required.
- **y_train** (*pd.Series*) – The target training data of length [n_samples]. Required for supervised learning tasks.
- **X_holdout** (*pd.DataFrame*) – The input holdout data of shape [n_samples, n_features].
- **y_holdout** (*pd.Series*) – The target holdout data of length [n_samples].
- **problem_type** (*str or ProblemTypes*) – Type of supervised learning problem. See evalml.problem_types.ProblemType.all_problem_types for a full list.
- **objective** (*str, ObjectiveBase*) – The objective to optimize for. Used to propose and rank pipelines, but not for optimizing each pipeline during fit-time. When set to 'auto', chooses: - LogLossBinary for binary classification problems, - LogLossMulticlass for multiclass classification problems, and - R2 for regression problems.
- **max_iterations** (*int*) – Maximum number of iterations to search. If max_iterations and max_time is not set, then max_iterations will default to max_iterations of 5.
- **max_time** (*int, str*) – Maximum time to search for pipelines. This will not start a new pipeline search after the duration has elapsed. If it is an integer, then the time will be in seconds. For strings, time can be specified as seconds, minutes, or hours.
- **patience** (*int*) – Number of iterations without improvement to stop search early. Must be positive. If None, early stopping is disabled. Defaults to None.
- **tolerance** (*float*) – Minimum percentage difference to qualify as score improvement for early stopping. Only applicable if patience is not None. Defaults to None.

- **allowed_component_graphs** (*dict*) – A dictionary of lists or ComponentGraphs indicating the component graphs allowed in the search. The format should follow { "Name_0": [list_of_components], "Name_1": ComponentGraph(...) }

  The default of None indicates all pipeline component graphs for this problem type are allowed. Setting this field will cause allowed_model_families to be ignored.

  e.g. allowed_component_graphs = { "My_Graph": ["Imputer", "One Hot Encoder", "Random Forest Classifier"] }

- **allowed_model_families** (*list(str, ModelFamily)*) – The model families to search. The default of None searches over all model families. Run evalml.pipelines.components.utils.allowed_model_families("binary") to see options. Change *binary* to *multiclass* or *regression* depending on the problem type. Note that if allowed_pipelines is provided, this parameter will be ignored.

- **features** (*list*) – List of features to run DFS on AutoML pipelines. Defaults to None. Features will only be computed if the columns used by the feature exist in the search input and if the feature itself is not in search input. If features is an empty list, the DFS Transformer will not be included in pipelines.

- **data_splitter** (*sklearn.model_selection.BaseCrossValidator*) – Data splitting method to use. Defaults to StratifiedKFold.

- **tuner_class** – The tuner class to use. Defaults to SKOptTuner.

- **optimize_thresholds** (*bool*) – Whether or not to optimize the binary pipeline threshold. Defaults to True.

- **start_iteration_callback** (*callable*) – Function called before each pipeline training iteration. Callback function takes three positional parameters: The pipeline instance and the AutoMLSearch object.

- **add_result_callback** (*callable*) – Function called after each pipeline training iteration. Callback function takes three positional parameters: A dictionary containing the training results for the new pipeline, an untrained_pipeline containing the parameters used during training, and the AutoMLSearch object.

- **error_callback** (*callable*) – Function called when *search()* errors and raises an Exception. Callback function takes three positional parameters: the Exception raised, the traceback, and the AutoMLSearch object. Must also accepts kwargs, so AutoMLSearch is able to pass along other appropriate parameters by default. Defaults to None, which will call *log_error_callback*.

- **additional_objectives** (*list*) – Custom set of objectives to score on. Will override default objectives for problem type if not empty.

- **alternate_thresholding_objective** (*str*) – The objective to use for thresholding binary classification pipelines if the main objective provided isn't tuneable. Defaults to F1.

- **random_seed** (*int*) – Seed for the random number generator. Defaults to 0.

- **n_jobs** (*int or None*) – Non-negative integer describing level of parallelism used for pipelines. None and 1 are equivalent. If set to -1, all CPUs are used. For n_jobs below -1, (n_cpus + 1 + n_jobs) are used.

- **ensembling** (*boolean*) – If True, runs ensembling in a separate batch after every allowed pipeline class has been iterated over. If the number of unique pipelines to search over per batch is one, ensembling will not run. Defaults to False.

- **max_batches** (*int*) – The maximum number of batches of pipelines to search. Parameters max_time, and max_iterations have precedence over stopping the search.

- **problem_configuration** (*dict, None*) – Additional parameters needed to configure the search. For example, in time series problems, values should be passed in for the time_index, gap, forecast_horizon, and max_delay variables.
- **train_best_pipeline** (*boolean*) – Whether or not to train the best pipeline before returning it. Defaults to True.
- **search_parameters** (*dict*) – A dict of the hyperparameter ranges or pipeline parameters used to iterate over during search. Keys should consist of the component names and values should specify a singular value/list for pipeline parameters, or skopt.Space for hyperparameter ranges. In the example below, the Imputer parameters would be passed to the hyperparameter ranges, and the Label Encoder parameters would be used as the component parameter.

  **e.g. search_parameters = { 'Imputer'** [{ 'numeric_impute_strategy': Categorical(['most_frequent', 'median']) },] 'Label Encoder': {'positive_label': True} }
- **sampler_method** (*str*) – The data sampling component to use in the pipelines if the problem type is classification and the target balance is smaller than the sampler_balanced_ratio. Either 'auto', which will use our preferred sampler for the data, 'Undersampler', 'Oversampler', or None. Defaults to 'auto'.
- **sampler_balanced_ratio** (*float*) – The minority:majority class ratio that we consider balanced, so a 1:4 ratio would be equal to 0.25. If the class balance is larger than this provided value, then we will not add a sampler since the data is then considered balanced. Overrides the *sampler_ratio* of the samplers. Defaults to 0.25.
- **allow_long_running_models** (*bool*) – Whether or not to allow longer-running models for large multiclass problems. If False and no pipelines, component graphs, or model families are provided, AutoMLSearch will not use Elastic Net or XGBoost when there are more than 75 multiclass targets and will not use CatBoost when there are more than 150 multiclass targets. Defaults to False.
- **_ensembling_split_size** (*float*) – The amount of the training data we'll set aside for training ensemble metalearners. Only used when ensembling is True. Must be between 0 and 1, exclusive. Defaults to 0.2
- **_pipelines_per_batch** (*int*) – The number of pipelines to train for every batch after the first one. The first batch will train a baseline pipline + one of each pipeline family allowed in the search.
- **automl_algorithm** (*str*) – The automl algorithm to use. Currently the two choices are 'iterative' and 'default'. Defaults to *default*.
- **engine** (*EngineBase or str*) – The engine instance used to evaluate pipelines. Dask or concurrent.futures engines can also be chosen by providing a string from the list ["sequential", "cf_threaded", "cf_process", "dask_threaded", "dask_process"]. If a parallel engine is selected this way, the maximum amount of parallelism, as determined by the engine, will be used. Defaults to "sequential".
- **verbose** (*boolean*) – Whether or not to display semi-real-time updates to stdout while search is running. Defaults to False.
- **timing** (*boolean*) – Whether or not to write pipeline search times to the logger. Defaults to False.
- **exclude_featurizers** (*list[str]*) – A list of featurizer components to exclude from the pipelines built by search. Valid options are "DatetimeFeaturizer", "EmailFeaturizer", "URLFeaturizer", "NaturalLanguageFeaturizer", "TimeSeriesFeaturizer"

- **holdout_set_size** (*float*) – The size of the holdout set that AutoML search will take for datasets larger than 500 rows. If set to 0, holdout set will not be taken regardless of number of rows. Must be between 0 and 1, exclusive. Defaults to 0.1.

**Methods**

| | |
|---|---|
| *add_to_rankings* | Fits and evaluates a given pipeline then adds the results to the automl rankings with the requirement that automl search has been run. |
| *best_pipeline* | Returns a trained instance of the best pipeline and parameters found during automl search. If *train_best_pipeline* is set to False, returns an untrained pipeline instance. |
| *close_engine* | Function to explicitly close the engine, client, parallel resources. |
| *describe_pipeline* | Describe a pipeline. |
| *full_rankings* | Returns a pandas.DataFrame with scoring results from all pipelines searched. |
| *get_ensembler_input_pipelines* | Returns a list of input pipeline IDs given an ensembler pipeline ID. |
| *get_pipeline* | Given the ID of a pipeline training result, returns an untrained instance of the specified pipeline initialized with the parameters used to train that pipeline during automl search. |
| *load* | Loads AutoML object at file path. |
| *plot* | Return an instance of the plot with the latest scores. |
| *rankings* | Returns a pandas.DataFrame with scoring results from the highest-scoring set of parameters used with each pipeline. |
| *results* | Class that allows access to a copy of the results from *automl_search*. |
| *save* | Saves AutoML object at file path. |
| *score_pipelines* | Score a list of pipelines on the given holdout data. |
| *search* | Find the best pipeline for the data set. |
| *train_pipelines* | Train a list of pipelines on the training data. |

**add_to_rankings**(*self*, *pipeline*)

Fits and evaluates a given pipeline then adds the results to the automl rankings with the requirement that automl search has been run.

**Parameters** **pipeline** (*PipelineBase*) – pipeline to train and evaluate.

**property best_pipeline**(*self*)

Returns a trained instance of the best pipeline and parameters found during automl search. If *train_best_pipeline* is set to False, returns an untrained pipeline instance.

**Returns** A trained instance of the best pipeline and parameters found during automl search. If *train_best_pipeline* is set to False, returns an untrained pipeline instance.

**Return type** PipelineBase

**Raises** **PipelineNotFoundError** – If this is called before .search() is called.

**close_engine**(*self*)

Function to explicitly close the engine, client, parallel resources.

**describe_pipeline**(*self*, *pipeline_id*, *return_dict=False*)

Describe a pipeline.

**Parameters**

- **pipeline_id** (*int*) – pipeline to describe
- **return_dict** (*bool*) – If True, return dictionary of information about pipeline. Defaults to False.

**Returns** Description of specified pipeline. Includes information such as type of pipeline components, problem, training time, cross validation, etc.

**Raises** **PipelineNotFoundError** – If pipeline_id is not a valid ID.

**property full_rankings**(*self*)

Returns a pandas.DataFrame with scoring results from all pipelines searched.

**get_ensembler_input_pipelines**(*self*, *ensemble_pipeline_id*)

Returns a list of input pipeline IDs given an ensembler pipeline ID.

**Parameters** **ensemble_pipeline_id** (*id*) – Ensemble pipeline ID to get input pipeline IDs from.

**Returns** A list of ensemble input pipeline IDs.

**Return type** list[int]

**Raises** **ValueError** – If *ensemble_pipeline_id* does not correspond to a valid ensemble pipeline ID.

**get_pipeline**(*self*, *pipeline_id*)

Given the ID of a pipeline training result, returns an untrained instance of the specified pipeline initialized with the parameters used to train that pipeline during automl search.

**Parameters** **pipeline_id** (*int*) – Pipeline to retrieve.

**Returns** Untrained pipeline instance associated with the provided ID.

**Return type** PipelineBase

**Raises** **PipelineNotFoundError** – if pipeline_id is not a valid ID.

**static load**(*file_path*, *pickle_type='cloudpickle'*)

Loads AutoML object at file path.

**Parameters**

- **file_path** (*str*) – Location to find file to load
- **pickle_type** (*{"pickle", "cloudpickle"}*) – The pickling library to use. Currently not used since the standard pickle library can handle cloudpickles.

**Returns** AutoSearchBase object

**property plot**(*self*)

Return an instance of the plot with the latest scores.

**property rankings**(*self*)

Returns a pandas.DataFrame with scoring results from the highest-scoring set of parameters used with each pipeline.

**property results**(*self*)

Class that allows access to a copy of the results from *automl_search*.

**Returns**

**Dictionary containing *pipeline_results*, a dict with results from each pipeline,** and *search_order*, a list describing the order the pipelines were searched.

**Return type** dict

**save**(*self*, *file_path*, *pickle_type='cloudpickle'*, *pickle_protocol=cloudpickle.DEFAULT_PROTOCOL*)

Saves AutoML object at file path.

**Parameters**

- **file_path** (*str*) – Location to save file.
- **pickle_type** (*{"pickle", "cloudpickle"}*) – The pickling library to use.
- **pickle_protocol** (*int*) – The pickle data stream format.

**Raises** **ValueError** – If pickle_type is not "pickle" or "cloudpickle".

**score_pipelines**(*self*, *pipelines*, *X_holdout*, *y_holdout*, *objectives*)

Score a list of pipelines on the given holdout data.

**Parameters**

- **pipelines** (*list[PipelineBase]*) – List of pipelines to train.
- **X_holdout** (*pd.DataFrame*) – Holdout features.
- **y_holdout** (*pd.Series*) – Holdout targets for scoring.
- **objectives** (*list[str], list[ObjectiveBase]*) – Objectives used for scoring.

**Returns** Dictionary keyed by pipeline name that maps to a dictionary of scores. Note that the any pipelines that error out during scoring will not be included in the dictionary but the exception and stacktrace will be displayed in the log.

**Return type** dict[str, Dict[str, float]]

**search**(*self*, *interactive_plot=True*)

Find the best pipeline for the data set.

**Parameters** **interactive_plot** (*boolean, True*) – Shows an iteration vs. score plot in Jupyter notebook. Disabled by default in non-Jupyter enviroments.

**Raises** **AutoMLSearchException** – If all pipelines in the current AutoML batch produced a score of np.nan on the primary objective.

**Returns** Dictionary keyed by batch number that maps to the timings for pipelines run in that batch, as well as the total time for each batch. Pipelines within a batch are labeled by pipeline name.

**Return type** Dict[int, Dict[str, Timestamp]]

**train_pipelines**(*self*, *pipelines*)

Train a list of pipelines on the training data.

This can be helpful for training pipelines once the search is complete.

**Parameters** **pipelines** (*list[PipelineBase]*) – List of pipelines to train.

**Returns** Dictionary keyed by pipeline name that maps to the fitted pipeline. Note that the any pipelines that error out during training will not be included in the dictionary but the exception and stacktrace will be displayed in the log.

**Return type** Dict[str, PipelineBase]

`evalml.search(X_train=None, y_train=None, problem_type=None, objective='auto', mode='fast', max_time=None, patience=None, tolerance=None, problem_configuration=None, n_splits=3, verbose=False, timing=False)`

Given data and configuration, run an automl search.

This method will run EvalML's default suite of data checks. If the data checks produce errors, the data check results will be returned before running the automl search. In that case we recommend you alter your data to address these errors and try again. This method is provided for convenience. If you'd like more control over when each of these steps is run, consider making calls directly to the various pieces like the data checks and AutoMLSearch, instead of using this method.

**Parameters**

- **X_train** (*pd.DataFrame*) – The input training data of shape [n_samples, n_features]. Required.
- **y_train** (*pd.Series*) – The target training data of length [n_samples]. Required for supervised learning tasks.
- **problem_type** (*str or ProblemTypes*) – Type of supervised learning problem. See evalml.problem_types.ProblemType.all_problem_types for a full list.
- **objective** (*str, ObjectiveBase*) – The objective to optimize for. Used to propose and rank pipelines, but not for optimizing each pipeline during fit-time. When set to 'auto', chooses: - LogLossBinary for binary classification problems, - LogLossMulticlass for multiclass classification problems, and - R2 for regression problems.
- **mode** (*str*) – mode for DefaultAlgorithm. There are two modes: fast and long, where fast is a subset of long. Please look at DefaultAlgorithm for more details.
- **max_time** (*int, str*) – Maximum time to search for pipelines. This will not start a new pipeline search after the duration has elapsed. If it is an integer, then the time will be in seconds. For strings, time can be specified as seconds, minutes, or hours.
- **patience** (*int*) – Number of iterations without improvement to stop search early. Must be positive. If None, early stopping is disabled. Defaults to None.
- **tolerance** (*float*) – Minimum percentage difference to qualify as score improvement for early stopping. Only applicable if patience is not None. Defaults to None.
- **problem_configuration** (*dict*) – Additional parameters needed to configure the search. For example, in time series problems, values should be passed in for the time_index, gap, forecast_horizon, and max_delay variables.
- **n_splits** (*int*) – Number of splits to use with the default data splitter.
- **verbose** (*boolean*) – Whether or not to display semi-real-time updates to stdout while search is running. Defaults to False.
- **timing** (*boolean*) – Whether or not to write pipeline search times to the logger. Defaults to False.

**Returns** The automl search object containing pipelines and rankings, and the results from running the data checks. If the data check results contain errors, automl search will not be run and an automl search object will not be returned.

**Return type** (*AutoMLSearch*, dict)

**Raises** `ValueError` – If search configuration is not valid.

`evalml.search_iterative(X_train=None, y_train=None, problem_type=None, objective='auto', problem_configuration=None, n_splits=3, timing=False, **kwargs)`

Given data and configuration, run an automl search.

This method will run EvalML's default suite of data checks. If the data checks produce errors, the data check results will be returned before running the automl search. In that case we recommend you alter your data to address these errors and try again. This method is provided for convenience. If you'd like more control over when each of these steps is run, consider making calls directly to the various pieces like the data checks and AutoMLSearch, instead of using this method.

**Parameters**

- **X_train** (*pd.DataFrame*) – The input training data of shape [n_samples, n_features]. Required.
- **y_train** (*pd.Series*) – The target training data of length [n_samples]. Required for supervised learning tasks.
- **problem_type** (*str or ProblemTypes*) – Type of supervised learning problem. See evalml.problem_types.ProblemType.all_problem_types for a full list.
- **objective** (*str, ObjectiveBase*) – The objective to optimize for. Used to propose and rank pipelines, but not for optimizing each pipeline during fit-time. When set to 'auto', chooses: - LogLossBinary for binary classification problems, - LogLossMulticlass for multiclass classification problems, and - R2 for regression problems.
- **problem_configuration** (*dict*) – Additional parameters needed to configure the search. For example, in time series problems, values should be passed in for the time_index, gap, forecast_horizon, and max_delay variables.
- **n_splits** (*int*) – Number of splits to use with the default data splitter.
- **timing** (*boolean*) – Whether or not to write pipeline search times to the logger. Defaults to False.
- ****kwargs** – Other keyword arguments which are provided will be passed to AutoMLSearch.

**Returns** the automl search object containing pipelines and rankings, and the results from running the data checks. If the data check results contain errors, automl search will not be run and an automl search object will not be returned.

**Return type** (*AutoMLSearch*, dict)

**Raises** `ValueError` – If the search configuration is invalid.

CHAPTER

SIX

# RELEASE NOTES

**Future Releases**

- Enhancements
- Fixes
- Changes
- Documentation Changes
- Testing Changes

**Warning: Breaking Changes**

**v0.74.0 Apr. 18, 2023**

- **Enhancements**
  - Saved computed additional_objectives computed during search to AutoML object #4141
  - Remove extra naive pipelines #4142
- **Fixes**
  - Fixed usage of codecov after uploader deprecation #4144
  - Fixed issue where prediction intervals were becoming NaNs due to index errors #4154
- **Changes**
  - Capped size of seasonal period used for determining whether to include STLDecomposer in pipelines #4147

**v0.73.0 Apr. 10, 2023**

- **Enhancements**
  - Allowed `InvalidTargetDataCheck` to return a `DROP_ROWS` `DataCheckActionOption` #4116
  - Implemented prediction intervals for non-time series native pipelines using the naïve method #4127
- **Changes**
  - Removed unnecessary logic from imputer components prior to nullable type handling #4038, #4043
  - Added calls to `_handle_nullable_types` in component fit, transform, and predict methods when needed #4046, #4043

  - Removed existing nullable type handling across AutoMLSearch to just use new handling #4085, #4043
  - Handled nullable type incompatibility in `Decomposer` #4105, :pr:`4043
  - Removed nullable type incompatibility handling for ARIMA and ExponentialSmoothingRegressor #4129
  - Changed the default value for `null_strategy` in `InvalidTargetDataCheck` to `drop` #4131
  - Pinned sktime version to 0.17.0 for nullable types support #4137
- **Testing Changes**
  - Fixed installation of prophet for linux nightly tests #4114

**v0.72.0 Mar. 27, 2023**

- **Enhancements**
  - Updated *pipeline.get_prediction_intervals()* to add trend prediction interval information from STL decomposer #4093
  - Added `method=all` support for `TargetLeakageDataCheck` #4106
- **Fixes**
  - Fixed ensemble pipelines not working with `generate_pipeline_example` #4102
- **Changes**
  - Pinned ipywidgets version under 8.0.5 #4097
  - Calculated partial dependence grid values for integer data by rounding instead of truncating fractional values #4096
- **Testing Changes**
  - Updated graphviz installation in GitHub workflows to fix windows nightlies #4088

**v0.71.0 Mar. 17, 2023***

- **Fixes**
  - Fixed error in `PipelineBase._supports_fast_permutation_importance` with stacked ensemble pipelines #4083

**v0.70.0 Mar. 16, 2023**

- **Changes**
  - Added Oversampler nullable type incompatibility in X #4068
  - Removed nullable handling from objective functions, `roc_curve`, and `correlation_matrix` #4072
  - Transitioned from `prophet-prebuilt` to `prophet` directly #4045

**v0.69.0 Mar. 15, 2023**

- **Enhancements**
  - Move black to regular dependency and use it for `generate_pipeline_code` #4005
  - Implement `generate_pipeline_example` #4023
  - Add new downcast utils for component-specific nullable type handling and begin implementation on objective and component base classes #4024

  - Add nullable type incompatibility properties to the components that need them #4031
  - Add `get_evalml_requirements_file` #4034
  - Pipelines with DFS Transformers will run fast permutation importance if DFS features pre-exist #4037
  - Add get_prediction_intervals() at the pipeline level #4052
- **Fixes**
  - Fixed `generate_pipeline_example` erroring out for pipelines with a `DFSTransformer` #4059
  - Remove nullable types handling for `OverSampler` #4064
- **Changes**
  - Uncapped `pmdarima` and updated minimum version #4027
  - Increase min catboost to 1.1.1 and xgboost to 1.7.0 to add nullable type support for those estimators #3996
  - Unpinned `networkx` and updated minimum version #4035
  - Increased `scikit-learn` version to 1.2.2 #4064
  - Capped max `holidays` version to 0.21 #4064
  - Stop allowing `knn` as a boolean impute strategy #4058
  - Capped `nbsphinx` at < 0.9.0 #4071
- **Testing Changes**
  - Use `release.yaml` for performance tests on merge to main #4007
  - Pin `github-action-check-linked-issues` at v1.4.5 #4042
  - Updated tests to support Woodwork's object dtype inference for numeric columns #4066
  - Updated `TargetLeakageDataCheck` tests to handle boolean targets properly #4066

**v0.68.0 Feb. 15, 2023**

- **Enhancements**
  - Integrated `determine_periodicity` into `AutoMLSearch` #3952
  - Removed frequency limitations for decomposition using the `STLDecomposer` #3952
- **Changes**
  - Remove requirements-parser requirement #3978
  - Updated the `SKOptTuner` to use a gradient boosting regressor for tuning instead of extra trees #3983
  - Unpinned sktime from below 1.2, increased minimum to 1.2.1 #3983
- **Testing Changes**
  - Add pull request check for linked issues to CI workflow #3970, #3980
  - Upgraded minimum *IPython* version to 8.10.0 #3987

**v0.67.0 Jan. 31, 2023**

- **Fixes**

  - Re-added `TimeSeriesPipeline.should_skip_featurization` to fix bug where data would get featurized unnecessarily #3964
  - Allow float categories to be passed into CatBoost estimators #3966
- **Changes**
  - Update pyproject.toml to correctly specify the data filepaths #3967
- **Documentation Changes**
  - Added demo for prediction intervals #3954

**v0.66.1 Jan. 26, 2023**

- **Fixes**
  - Updated `LabelEncoder` to store the original typing information #3960
  - Fixed bug where all-null `BooleanNullable` columns would break the imputer during transform #3959

**v0.66.0 Jan. 24, 2023**

- **Enhancements**
  - Improved decomposer `determine_periodicity` functionality for better period guesses #3912
  - Added `dates_needed_for_prediction` for time series pipelines #3906
  - Added `RFClassifierRFESelector` and `RFRegressorRFESelector` components for feature selection using recursive feature elimination #3934
  - Added `dates_needed_for_prediction_range` for time series pipelines #3941
- **Fixes**
  - Fixed `set_period()` not updating decomposer parameters #3932
  - Removed second identical batch for time series problems in `DefaultAlgorithm` #3936
  - Fix install command for alteryx-open-src-update-checker #3940
  - Fixed non-prophet case of `test_components_can_be_used_for_partial_dependence_fast_mode` #3949
- **Changes**
  - Updated `PolynomialDecomposer` to work with sktime v0.15.1 #3930
  - Add ruff and use pyproject.toml (move away from setup.cfg) #3928
  - Pinned *category-encoders* ` to 2.5.1.post0 #3933
  - Remove requirements-parser and tomli from core requirements #3948

**v0.65.0 Jan. 3, 2023**

- **Enhancements**
  - Added the ability to retrieve prediction intervals for estimators that support time series regression #3876
  - Added utils to handle the logic for threshold tuning objective and resplitting data #3888
  - Integrated `OrdinalEncoder` into AutoMLSearch #3765
- **Fixes**
  - Fixed ARIMA not accounting for gap in prediction from end of training data #3884

  - Fixed `DefaultAlgorithm` adding an extra `OneHotEncoder` when a categorical column is not selected #3914

- **Changes**
  - Added a threshold to `DateTimeFormatDataCheck` to account for too many duplicate or nan values #3883
  - Changed treatment of `Boolean` columns for `SimpleImputer` and `ClassImbalanceDataCheck` to be compatible with new Woodwork inference #3892
  - Split decomposer `seasonal_period` parameter into `seasonal_smoother` and `period` parameters #3896
  - Excluded catboost from the broken link checking workflow due to 403 errors #3899
  - Pinned scikit-learn version below 1.2.0 #3901
  - Cast newly created one hot encoded columns as `bool` dtype #3913
- **Documentation Changes**
  - Hid non-essential warning messages in time series docs #3890
- Testing Changes

**v0.64.0 Dec. 8, 2022**

- Enhancements
- **Fixes**
  - Allowed the DFS Transformer to calculate feature values for Features with a `dataframe_name` that is not `"X"` #3873
  - Stopped passing full list of DFS Transformer features into cloned pipeline in partial dependence fast mode #3875
- **Changes**
  - Update leaderboard names to show *ranking_score* instead of *validation_score* #3878
  - Remove Int64Index after Pandas 1.5 Upgrade #3825
  - Reduced the threshold for setting `use_covariates` to False for ARIMA models in AutoMLSearch #3868
  - Pinned woodwork version at <=0.19.0 #3871
  - Updated minimum Pandas version to 1.5.0 #3808
  - Remove dsherry from automated dependency update reviews and added tamargrey #3870
- Documentation Changes
- Testing Changes

**v0.63.0 Nov. 23, 2022**

- **Enhancements**
  - Added fast mode to partial dependence #3753
  - Added the ability to serialize featuretools features into time series pipelines #3836
- **Fixes**
  - Fixed `TimeSeriesFeaturizer` potentially selecting lags outside of feature engineering window #3773

  - Fixed bug where `TimeSeriesFeaturizer` could not encode Ordinal columns with non numeric categories #3812
  - Updated demo dataset links to point to new endpoint #3826
  - Updated `STLDecomposer` to infer the time index frequency if it's not present #3829
  - Updated `_drop_time_index` to move the time index from X to both `X.index` and `y.index` #3829
  - Fixed bug where engineered features lost their origin attribute in partial dependence, causing it to fail #3830
  - Fixed bug where partial dependence's fast mode handling for the DFS Transformer wouldn't work with multi output features #3830
  - Allowed target to be present and ignored in partial dependence's DFS Transformer fast mode handling #3830
- **Changes**
  - Consolidated decomposition frequency validation logic to `Decomposer` class #3811
  - Removed Featuretools version upper bound and prevent Woodwork 0.20.0 from being installed #3813
  - Updated min Featuretools version to 0.16.0, min nlp-primitives version to 2.9.0 and min Dask version to 2022.2.0 #3823
  - Rename issue templates config.yaml to config.yml #3844
  - Reverted change adding a `should_skip_featurization` flag to time series pipelines #3862
- **Documentation Changes**
  - Added information about STL Decomposition to the time series docs #3835
  - Removed RTD failure on warnings #3864

**v0.62.0 Nov. 01, 2022**

- **Fixes**
  - Fixed bug with datetime conversion in `get_time_index` #3792
  - Fixed bug where invalid anchored or offset frequencies were including the `STLDecomposer` in pipelines #3794
  - Fixed bug where irregular datetime frequencies were causing errors in `make_pipeline` #3800
- **Changes**
  - Capped dask at < 2022.10.1 #3797
  - Uncapped dask and excluded 2022.10.1 from viable versions #3803
  - Removed all references to XGBoost's deprecated `_use_label_encoder` argument #3805
  - Capped featuretools at < 1.17.0 #3805
  - Capped woodwork at < 0.21.0 #3805

**v0.61.1 Oct. 27, 2022**

- **Fixes**
  - Fixed bug where `TimeSeriesBaselinePipeline` wouldn't preserve index name of input features #3788

  - Fixed bug in `TimeSeriesBaselinePipeline` referencing a static string instead of time index var #3788

- **Documentation Changes**
  - Updated Release Notes #3788

**v0.61.0 Oct. 25, 2022**

- **Enhancements**
  - Added the STL Decomposer #3741
  - Integrated STLDecomposer into AutoMLSearch for time series regression problems #3781
  - Brought the PolynomialDecomposer up to parity with STLDecomposer #3768
- **Changes**
  - Cap Featuretools at < 1.15.0 #3775
  - Remove Featuretools upper bound restriction and fix nlp-primitives import statements #3778

**v0.60.0 Oct. 19, 2022**

- **Enhancements**
  - Add forecast functions to time series regression pipeline #3742
- **Fixes**
  - Fix to allow `IDColumnsDataCheck` to work with `IntegerNullable` inputs #3740
  - Fixed datasets name for main performance tests #3743
- **Changes**
  - Use Woodwork's `dependence_dict` method to calculate for `TargetLeakageDataCheck` #3728
- Documentation Changes
- Testing Changes

> **Warning:**
>
> **Breaking Changes**
>
> - `TargetLeakageDataCheck` now uses argument `mutual_info` rather than `mutual` #3728

**v0.59.0 Sept. 27, 2022**

- **Enhancements**
  - Enhanced Decomposer with `determine_periodicity` function to automatically determine periodicity of seasonal target. #3729
  - Enhanced Decomposer with `set_seasonal_period` function to set a `Decomposer` object's seasonal period automatically. #3729
  - Added `OrdinalEncoder` component #3736
- **Fixes**
  - Fixed holdout warning message showing when using default parameters #3727
  - Fixed bug in Oversampler where categorical dtypes would fail #3732
- **Changes**

  - Automatic sorting of the `time_index` prior to running `DataChecks` has been disabled #3723
- Documentation Changes
- **Testing Changes**
  - Update job to use new looking glass report command #3733

**v0.58.0 Sept. 20, 2022**

- **Enhancements**
  - Defined *get_trend_df()* for PolynomialDecomposer to allow decomposition of target data into trend, seasonality and residual. #3720
  - Updated to run with Woodwork >= 0.18.0 #3700
  - Pass time index column to time series native estimators but drop otherwise #3691
  - Added `errors` attribute to `AutoMLSearch` for useful debugging #3702
- **Fixes**
  - Removed multiple samplers occurring in pipelines generated by `DefaultAlgorithm` #3696
  - Fix search order changing when using `DefaultAlgorithm` #3704
- **Changes**
  - Bumped up minimum version of sktime to 0.12.0. #3720
  - Added abstract Decomposer class as a parent to PolynomialDecomposer to support additional decomposers. #3720
  - Pinned `pmdarima` < 2.0.0 #3679
  - Added support for using `downcast_nullable_types` with Series as well as DataFrames #3697
  - Added distinction between ranking and optimization objectives #3721
- Documentation Changes
- **Testing Changes**
  - Updated pytest fixtures and brittle test files to explicitly set woodwork typing information #3697
  - Added github workflow to run looking glass performance tests on merge to main #3690
  - Fixed looking glass performance test script #3715
  - Remove commit message from looking glass slack message #3719

**v0.57.0 Sept. 6, 2022**

- **Enhancements**
  - Added `KNNImputer` class and created new knn parameter for Imputer #3662
- **Fixes**
  - `IDColumnsDataCheck` now only returns an action code to set the first column as the primary key if it contains unique values #3639
  - `IDColumnsDataCheck` now can handle primary key columns containing "integer" values that are of the double type #3683
  - Added support for BooleanNullable columns in EvalML pipelines and imputer #3678
  - Updated StandardScaler to only apply to numeric columns #3686

- **Changes**
  - Unpinned sktime to allow for version 0.13.2 #3685
  - Pinned `pmdarima` < 2.0.0 #3679

**v0.56.1 Aug. 19, 2022**

- **Fixes**
  - `IDColumnsDataCheck` now only returns an action code to set the first column as the primary key if it contains unique values #3639
  - Reverted the `make_pipeline` changes that conditionally included the imputers #3672

**v0.56.0 Aug. 15, 2022**

- **Enhancements**
  - Add CI testing environment in Mac for install workflow #3646
  - Updated `make_pipeline` to only include the Imputer in pipelines if NaNs exist in the data #3657
  - Updated to run with Woodwork >= 0.17.2 #3626
  - Add `exclude_featurizers` parameter to `AutoMLSearch` to specify featurizers that should be excluded from all pipelines #3631
  - Add `fit_transform` method to pipelines and component graphs #3640
  - Changed default value of data splitting for time series problem holdout set evaluation #3650
- **Fixes**
  - Reverted the Woodwork 0.17.x compatibility work due to performance regression #3664
- **Changes**
  - Disable holdout set in AutoML search by default #3659
  - Pinned `sktime` at >=0.7.0,<0.13.1 due to slowdowns with time series modeling #3658
  - Added additional testing support for Python 3.10 #3609
- **Documentation Changes**
  - Updated broken link checker to exclude stackoverflow domain #3633
  - Add instructions to add new users to evalml-core-feedstock #3636

**v0.55.0 July. 24, 2022**

- **Enhancements**
  - Increased the amount of logical type information passed to Woodwork when calling `ww.init()` in transformers #3604
  - Added ability to log how long each batch and pipeline take in `automl.search()` #3577
  - Added the option to set the `sp` parameter for ARIMA models #3597
  - Updated the CV split size of time series problems to match forecast horizon for improved performance #3616
  - Added holdout set evaluation as part of AutoML search and pipeline ranking #3499
  - Added Dockerfile.arm and .dockerignore for python version and M1 testing #3609
  - Added `test_gen_utils::in_container_arm64()` fixture #3609

- **Fixes**
  - Fixed iterative graphs not appearing in documentation #3592
  - Updated the `load_diabetes()` method to account for scikit-learn 1.1.1 changes to the dataset #3591
  - Capped woodwork version at < 0.17.0 #3612
  - Bump minimum scikit-optimize version to 0.9.0 *:pr: `3614*
  - Invalid target data checks involving regression and unsupported data types now produce a different `DataCheckMessageCode` #3630
  - Updated `test_data_checks.py::test_data_checks_raises_value_errors_on_init` - more lenient text check #3609
- **Changes**
  - Add pre-commit hooks for linting #3608
  - Implemented a lower threshold and window size for the `TimeSeriesRegularizer` and `DatetimeFormatDataCheck` #3627
  - Updated `IDColumnsDataCheck` to return an action to set the first column as the primary key if it is identified as an ID column #3634
- Documentation Changes
- **Testing Changes**
  - Pinned GraphViz version for Windows CI Test #3596
  - Removed skipping of PolynomialDecomposer tests for Python 3.9 envs. #3720
  - Removed `pytest.mark.skip_if_39` pytest marker #3602 #3607
  - Updated pytest==7.1.2 #3609
  - Added Dockerfile.arm and .dockerignore for python version and M1 testing #3609
  - Added `test_gen_utils::in_container_arm64()` fixture #3609

**Warning:**

**Breaking Changes**

- Refactored test cases that iterate over all components to use `pytest.mark.parametrise` and changed the corresponding `if...continue` blocks to `pytest.mark.xfail` #3622

**v0.54.0 June. 23, 2022**

- **Fixes**
  - Updated the Imputer and SimpleImputer to work with scikit-learn 1.1.1. #3525
  - Bumped the minimum versions of scikit-learn to 1.1.1 and imbalanced-learn to 0.9.1. #3525
  - Added a clearer error message when `describe` is called on an un-instantiated ComponentGraph #3569
  - Added a clearer error message when time series' `predict` is called with its X_train or y_train parameter set as None #3579
- **Changes**

  - Don't pass `time_index` as kwargs to sktime ARIMA implementation for compatibility with latest version #3564
  - Remove incompatible `nlp-primitives` version 2.6.0 from accepted dependency versions #3572, #3574
  - Updated evalml authors #3581
- **Documentation Changes**
  - Fix typo in `long_description` field in `setup.cfg` #3553
  - Update install page to remove Python 3.7 mention #3567

**v0.53.1 June. 9, 2022**

- **Changes**
  - Set the development status to `4 - Beta` in `setup.cfg` #3550

**v0.53.0 June. 9, 2022**

- **Enhancements**
  - Pass `n_jobs` to default algorithm #3548
- **Fixes**
  - Fixed github workflows for featuretools and woodwork to test their main branch against evalml. #3517
  - Supress warnings in `TargetEncoder` raised by a coming change to default parameters #3540
  - Fixed bug where schema was not being preserved in column renaming for XGBoost and LightGBM models #3496
- **Changes**
  - Transitioned to use pyproject.toml and setup.cfg away from setup.py #3494, #3536
- **Documentation Changes**
  - Updated the Time Series User Guide page to include known-in-advance features and fix typos #3521
  - Add slack and stackoverflow icon to footer #3528
  - Add install instructions for M1 Mac #3543
- **Testing Changes**
  - Rename yml to yaml for GitHub Actions #3522
  - Remove `noncore_dependency` pytest marker #3541
  - Changed `test_smotenc_category_features` to use valid postal code values in response to new woodwork type validation #3544

**v0.52.0 May. 12, 2022**

- **Changes**
  - Added github workflows for featuretools and woodwork to test their main branch against evalml. #3504
  - Added pmdarima to conda recipe. #3505
  - Added a threshold for `NullDataCheck` before a warning is issued for null values #3507

  - Changed `NoVarianceDataCheck` to only output warnings #3506
  - Reverted XGBoost Classifier/Regressor patch for all boolean columns needing to be converted to int. #3503
  - Updated `roc_curve()` and `conf_matrix()` to work with IntegerNullable and BooleanNullable types. #3465
  - Changed `ComponentGraph._transform_features` to raise a `PipelineError` instead of a `ValueError`. This is not a breaking change because `PipelineError` is a subclass of `ValueError`. #3497
  - Capped `sklearn` at version 1.1.0 #3518
- **Documentation Changes**
  - Updated to install prophet extras in Read the Docs. #3509
- **Testing Changes**
  - Moved vowpal wabbit in test recipe to `evalml` package from `evalml-core` #3502

**v0.51.0 Apr. 28, 2022**

- **Enhancements**
  - Updated `make_pipeline_from_data_check_output` to work with time series problems. #3454
- **Fixes**
  - Changed `PipelineBase.graph_json()` to return a python dictionary and renamed as `graph_dict()`#3463
- **Changes**
  - Added `vowpalwabbit` to local recipe and remove `is_using_conda` pytest skip markers from relevant tests #3481
- **Documentation Changes**
  - Fixed broken link in contributing guide #3464
  - Improved development instructions #3468
  - Added the `TimeSeriesRegularizer` and `TimeSeriesImputer` to the timeseries section of the User Guide #3473
  - Updated OSS slack link #3487
  - Fix rendering of model understanding plotly charts in docs #3460
- **Testing Changes**
  - Updated unit tests to support woodwork 0.16.2 #3482
  - Fix some unit tests after vowpal wabbit got added to conda recipe #3486

**Warning:**

**Breaking Changes**

- Renamed `PipelineBase.graph_json()` to `PipelineBase.graph_dict()` #3463
- Minimum supported woodwork version is now 0.16.2 #3482

**v0.50.0 Apr. 12, 2022**

- **Enhancements**
  - Added `TimeSeriesImputer` component #3374
  - Replaced `pipeline_parameters` and `custom_hyperparameters` with `search_parameters` in `AutoMLSearch` #3373, #3427
  - Added `TimeSeriesRegularizer` to smooth uninferrable date ranges for time series problems #3376
  - Enabled ensembling as a parameter for `DefaultAlgorithm` #3435, #3444
- **Fixes**
  - Fix `DefaultAlgorithm` not handling Email and URL features #3419
  - Added test to ensure `LabelEncoder` parameters preserved during `AutoMLSearch` #3326
- **Changes**
  - Updated `DateTimeFormatDataCheck` to use woodwork's `infer_frequency` function #3425
  - Renamed `graphs.py` to `visualizations.py` #3439
- **Documentation Changes**
  - Updated the model understanding section of the user guide to include missing functions #3446
  - Rearranged the user guide model understanding page for easier navigation #3457
  - Update README text to Alteryx #3462

**Warning:**

**Breaking Changes**

- Renamed `graphs.py` to `visualizations.py` #3439
- Replaced `pipeline_parameters` and `custom_hyperparameters` with `search_parameters` in `AutoMLSearch` #3373

**v0.49.0 Mar. 31, 2022**

- **Enhancements**
  - Added `use_covariates` parameter to `ARIMARegressor` #3407
  - `AutoMLSearch` will set `use_covariates` to `False` for ARIMA when dataset is large #3407
  - Add ability to retrieve logical types to a component in the graph via `get_component_input_logical_types` #3428
  - Add ability to get logical types passed to the last component via `last_component_input_logical_types` #3428
- **Fixes**
  - Fix conda build after PR *3407* #3429
- **Changes**
  - Moved model understanding metrics from `graph.py` into a separate file #3417
  - Unpin `click` dependency #3420
  - For `IterativeAlgorithm`, put time series algorithms first #3407

  - Use `prophet-prebuilt` to install prophet in extras #3407

> **Warning:**
>
> **Breaking Changes**
>
> - Moved model understanding metrics from `graph.py` to `metrics.py` #3417

**v0.48.0 Mar. 25, 2022**

- **Enhancements**
  - Add support for oversampling in time series classification problems #3387
- **Fixes**
  - Fixed `TimeSeriesFeaturizer` to make it deterministic when creating and choosing columns #3384
  - Fixed bug where Email/URL features with missing values would cause the imputer to error out #3388
- **Changes**
  - Update maintainers to add Frank #3382
  - Allow woodwork version 0.14.0 to be installed #3381
  - Moved partial dependence functions from `graph.py` to a separate file #3404
  - Pin `click` at `8.0.4` due to incompatibility with `black` #3413
- **Documentation Changes**
  - Added automl user guide section covering search algorithms #3394
  - Updated broken links and automated broken link detection #3398
  - Upgraded nbconvert #3402, #3411
- **Testing Changes**
  - Updated scheduled workflows to only run on Alteryx owned repos (#3395)
  - Exclude documentation versions other than latest from broken link check #3401

> **Warning:**
>
> **Breaking Changes**
>
> - Moved partial dependence functions from `graph.py` to `partial_dependence.py` #3404

**v0.47.0 Mar. 16, 2022**

- **Enhancements**
  - Added `TimeSeriesFeaturizer` into ARIMA-based pipelines #3313
  - Added caching capability for ensemble training during `AutoMLSearch` #3257
  - Added new error code for zero unique values in `NoVarianceDataCheck` #3372
- **Fixes**
  - Fixed `get_pipelines` to reset pipeline threshold for binary cases #3360

- **Changes**
  - Update maintainers #3365
  - Revert pandas 1.3.0 compatibility patch #3378
- **Documentation Changes**
  - Fixed documentation links to point to correct pages #3358
- **Testing Changes**
  - Checkout main branch in build_conda_pkg job #3375

**v0.46.0 Mar. 03, 2022**

- **Enhancements**
  - Added `test_size` parameter to `ClassImbalanceDataCheck` #3341
  - Make target optional for `NoVarianceDataCheck` #3339
- **Changes**
  - Removed `python_version<3.9` environment marker from sktime dependency #3332
  - Updated `DatetimeFormatDataCheck` to return all messages and not return early if NaNs are detected #3354
- **Documentation Changes**
  - Added in-line tabs and copy-paste functionality to documentation, overhauled Install page #3353

**v0.45.0 Feb. 17, 2022**

- **Enhancements**
  - Added support for pandas >= 1.4.0 #3324
  - Standardized feature importance for estimators #3305
  - Replaced usage of private method with Woodwork's public `get_subset_schema` method #3325
- **Changes**
  - Added an `is_cv` property to the datasplitters used #3297
  - Changed SimpleImputer to ignore Natural Language columns #3324
  - Added drop NaN component to some time series pipelines #3310
- **Documentation Changes**
  - Update README.md with Alteryx link (#3319)
  - Added formatting to the AutoML user guide to shorten result outputs #3328
- **Testing Changes**
  - Add auto approve dependency workflow schedule for every 30 mins #3312

**v0.44.0 Feb. 04, 2022**

- **Enhancements**
  - Updated `DefaultAlgorithm` to also limit estimator usage for long-running multiclass problems #3099
  - Added `make_pipeline_from_data_check_output()` utility method #3277

  - Updated `AutoMLSearch` to use `DefaultAlgorithm` as the default automl algorithm #3261, #3304
  - Added more specific data check errors to `DatetimeFormatDataCheck` #3288
  - Added `features` as a parameter for `AutoMLSearch` and add `DFSTransformer` to pipelines when `features` are present #3309
- **Fixes**
  - Updated the binary classification pipeline's `optimize_thresholds` method to use Nelder-Mead #3280
  - Fixed bug where feature importance on time series pipelines only showed 0 for time index #3285
- **Changes**
  - Removed `DateTimeNaNDataCheck` and `NaturalLanguageNaNDataCheck` in favor of `NullDataCheck` #3260
  - Drop support for Python 3.7 #3291
  - Updated minimum version of `woodwork` to `v0.12.0` #3290
- **Documentation Changes**
  - Update documentation and docstring for *validate_holdout_datasets* for time series problems #3278
  - Fixed mistake in documentation where wrong objective was used for calculating percent-better-than-baseline #3285

**Warning:**

**Breaking Changes**

- Removed `DateTimeNaNDataCheck` and `NaturalLanguageNaNDataCheck` in favor of `NullDataCheck` #3260
- Dropped support for Python 3.7 #3291

**v0.43.0 Jan. 25, 2022**

- **Enhancements**
  - Updated new `NullDataCheck` to return a warning and suggest an action to impute columns with null values #3197
  - Updated `make_pipeline_from_actions` to handle null column imputation #3237
  - Updated data check actions API to return options instead of actions and add functionality to suggest and take action on columns with null values #3182
- **Fixes**
  - Fixed categorical data leaking into non-categorical sub-pipelines in `DefaultAlgorithm` #3209
  - Fixed Python 3.9 installation for prophet by updating `pmdarima` version in requirements #3268
  - Allowed DateTime columns to pass through PerColumnImputer without breaking #3267
- **Changes**
  - Updated `DataCheck` `validate()` output to return a dictionary instead of list for actions #3142
  - Updated `DataCheck` `validate()` API to use the new `DataCheckActionOption` class instead of `DataCheckAction` #3152

  - Uncapped numba version and removed it from requirements #3263
  - Renamed `HighlyNullDataCheck` to `NullDataCheck` #3197
  - Updated data check `validate()` output to return a list of warnings and errors instead of a dictionary #3244
  - Capped `pandas` at < 1.4.0 #3274
- **Testing Changes**
  - Bumped minimum `IPython` version to 7.16.3 in `test-requirements.txt` based on dependabot feedback #3269

**Warning:**

**Breaking Changes**

- Renamed `HighlyNullDataCheck` to `NullDataCheck` #3197
- Updated data check `validate()` output to return a list of warnings and errors instead of a dictionary. See the Data Check or Data Check Actions pages (under User Guide) for examples. #3244
- Removed `impute_all` and `default_impute_strategy` parameters from the `PerColumnImputer` #3267
- Updated `PerColumnImputer` such that columns not specified in `impute_strategies` dict will not be imputed anymore #3267

**v0.42.0 Jan. 18, 2022**

- **Enhancements**
  - Required the separation of training and test data by `gap` + 1 units to be verified by `time_index` for time series problems #3208
  - Added support for boolean features for `ARIMARegressor` #3187
  - Updated dependency bot workflow to remove outdated description and add new configuration to delete branches automatically #3212
  - Added `n_obs` and `n_splits` to `TimeSeriesParametersDataCheck` error details #3246
- **Fixes**
  - Fixed classification pipelines to only accept target data with the appropriate number of classes #3185
  - Added support for time series in `DefaultAlgorithm` #3177
  - Standardized names of featurization components #3192
  - Removed empty cell in text_input.ipynb #3234
  - Removed potential prediction explanations failure when pipelines predicted a class with probability 1 #3221
  - Dropped NaNs before partial dependence grid generation #3235
  - Allowed prediction explanations to be json-serializable #3262
  - Fixed bug where `InvalidTargetDataCheck` would not check time series regression targets #3251
  - Fixed bug in `are_datasets_separated_by_gap_time_index` #3256

- **Changes**
  - Raised lowest compatible numpy version to 1.21.0 to address security concerns #3207
  - Changed the default objective to `MedianAE` from `R2` for time series regression #3205
  - Removed all-nan Unknown to Double logical conversion in `infer_feature_types` #3196
  - Checking the validity of holdout data for time series problems can be performed by calling `pipelines.utils.validate_holdout_datasets` prior to calling `predict` #3208
- **Testing Changes**
  - Update auto approve workflow trigger and delete branch after merge #3265

> **Warning:**
>
> **Breaking Changes**
>
> - Renamed `DateTime Featurizer Component` to `DateTime Featurizer` and `Natural Language Featurization Component` to `Natural Language Featurizer` #3192

**v0.41.0 Jan. 06, 2022**

- **Enhancements**
  - Added string support for DataCheckActionCode #3167
  - Added `DataCheckActionOption` class #3134
  - Add issue templates for bugs, feature requests and documentation improvements for GitHub #3199
- **Fixes**
  - Fix bug where prediction explanations `class_name` was shown as float for boolean targets #3179
  - Fixed bug in nightly linux tests #3189
- **Changes**
  - Removed usage of scikit-learn's `LabelEncoder` in favor of ours #3161
  - Removed nullable types checking from `infer_feature_types` #3156
  - Fixed `mean_cv_data` and `validation_score` values in AutoMLSearch.rankings to reflect cv score or `NaN` when appropriate #3162
- **Testing Changes**
  - Updated tests to use new pipeline API instead of defining custom pipeline classes #3172
  - Add workflow to auto-merge dependency PRs if status checks pass #3184

**v0.40.0 Dec. 22, 2021**

- **Enhancements**
  - Added `TimeSeriesSplittingDataCheck` to `DefaultDataChecks` to verify adequate class representation in time series classification problems #3141
  - Added the ability to accept serialized features and skip computation in `DFSTransformer` #3106
  - Added support for known-in-advance features #3149
  - Added Holt-Winters `ExponentialSmoothingRegressor` for time series regression problems #3157

  - Required the separation of training and test data by `gap` + 1 units to be verified by `time_index` for time series problems #3160

- **Fixes**
  - Fixed error caused when tuning threshold for time series binary classification #3140

- **Changes**
  - `TimeSeriesParametersDataCheck` was added to `DefaultDataChecks` for time series problems #3139
  - Renamed `date_index` to `time_index` in `problem_configuration` for time series problems #3137
  - Updated `nlp-primitives` minimum version to 2.1.0 #3166
  - Updated minimum version of `woodwork` to v0.11.0 #3171
  - Revert *3160* until uninferrable frequency can be addressed earlier in the process #3198

- **Documentation Changes**
  - Added comments to provide clarity on doctests #3155

- **Testing Changes**
  - Parameterized tests in `test_datasets.py` #3145

> **Warning:**
>
> **Breaking Changes**
>
> - Renamed `date_index` to `time_index` in `problem_configuration` for time series problems #3137

**v0.39.0 Dec. 9, 2021**

- **Enhancements**
  - Renamed `DelayedFeatureTransformer` to `TimeSeriesFeaturizer` and enhanced it to compute rolling features #3028
  - Added ability to impute only specific columns in `PerColumnImputer` #3123
  - Added `TimeSeriesParametersDataCheck` to verify the time series parameters are valid given the number of splits in cross validation #3111

- **Fixes**
  - Default parameters for `RFRegressorSelectFromModel` and `RFClassifierSelectFromModel` has been fixed to avoid selecting all features #3110

- **Changes**
  - Removed reliance on a datetime index for `ARIMARegressor` and `ProphetRegressor` #3104
  - Included target leakage check when fitting `ARIMARegressor` to account for the lack of `TimeSeriesFeaturizer` in `ARIMARegressor` based pipelines #3104
  - Cleaned up and refactored `InvalidTargetDataCheck` implementation and docstring #3122
  - Removed indices information from the output of `HighlyNullDataCheck`'s `validate()` method #3092
  - Added `ReplaceNullableTypes` component to prepare for handling pandas nullable types. #3090

  - Updated `make_pipeline` for handling pandas nullable types in preprocessing pipeline. #3129
  - Removed unused `EnsembleMissingPipelinesError` exception definition #3131
- **Testing Changes**
  - Refactored tests to avoid using `importorskip` #3126
  - Added `skip_during_conda` test marker to skip tests that are not supposed to run during conda build #3127
  - Added `skip_if_39` test marker to skip tests that are not supposed to run during python 3.9 #3133

**Warning:**

**Breaking Changes**

- Renamed `DelayedFeatureTransformer` to `TimeSeriesFeaturizer` #3028
- `ProphetRegressor` now requires a datetime column in `X` represented by the `date_index` parameter #3104
- Renamed module `evalml.data_checks.invalid_target_data_check` to `evalml.data_checks.invalid_targets_data_check` #3122
- Removed unused `EnsembleMissingPipelinesError` exception definition #3131

**v0.38.0 Nov. 27, 2021**

- **Enhancements**
  - Added `data_check_name` attribute to the data check action class #3034
  - Added `NumWords` and `NumCharacters` primitives to `TextFeaturizer` and renamed `TextFeaturizer` to ``NaturalLanguageFeaturizer` #3030
  - Added support for `scikit-learn > 1.0.0` #3051
  - Required the `date_index` parameter to be specified for time series problems in `AutoMLSearch` #3041
  - Allowed time series pipelines to predict on test datasets whose length is less than or equal to the `forecast_horizon`. Also allowed the test set index to start at 0. #3071
  - Enabled time series pipeline to predict on data with features that are not known-in-advanced #3094
- **Fixes**
  - Added in error message when fit and predict/predict_proba data types are different #3036
  - Fixed bug where ensembling components could not get converted to JSON format #3049
  - Fixed bug where components with tuned integer hyperparameters could not get converted to JSON format #3049
  - Fixed bug where force plots were not displaying correct feature values #3044
  - Included confusion matrix at the pipeline threshold for `find_confusion_matrix_per_threshold` #3080
  - Fixed bug where One Hot Encoder would error out if a non-categorical feature had a missing value #3083
  - Fixed bug where features created from categorical columns by `Delayed Feature Transformer` would be inferred as categorical #3083

- **Changes**
  - Delete `predict_uses_y` estimator attribute #3069
  - Change `DateTimeFeaturizer` to use corresponding Featuretools primitives #3081
  - Updated `TargetDistributionDataCheck` to return metadata details as floats rather strings #3085
  - Removed dependency on `psutil` package #3093
- **Documentation Changes**
  - Updated docs to use data check action methods rather than manually cleaning data #3050
- **Testing Changes**
  - Updated integration tests to use `make_pipeline_from_actions` instead of private method #3047

**Warning:**

**Breaking Changes**

- Added `data_check_name` attribute to the data check action class #3034
- Renamed `TextFeaturizer` to ``NaturalLanguageFeaturizer` #3030
- Updated the `Pipeline.graph_json` function to return a dictionary of "from" and "to" edges instead of tuples #3049
- Delete `predict_uses_y` estimator attribute #3069
- Changed time series problems in `AutoMLSearch` to need a not-`None` `date_index` #3041
- Changed the `DelayedFeatureTransformer` to throw a `ValueError` during fit if the `date_index` is `None` #3041
- Passing `X=None` to `DelayedFeatureTransformer` is deprecated #3041

**v0.37.0 Nov. 9, 2021**

- **Enhancements**
  - Added `find_confusion_matrix_per_threshold` to Model Understanding #2972
  - Limit computationally-intensive models during `AutoMLSearch` for certain multiclass problems, allow for opt-in with parameter `allow_long_running_models` #2982
  - Added support for stacked ensemble pipelines to prediction explanations module #2971
  - Added integration tests for data checks and data checks actions workflow #2883
  - Added a change in pipeline structure to handle categorical columns separately for pipelines in `DefaultAlgorithm` #2986
  - Added an algorithm to `DelayedFeatureTransformer` to select better lags #3005
  - Added test to ensure pickling pipelines preserves thresholds #3027
  - Added AutoML function to access ensemble pipeline's input pipelines IDs #3011
  - Added ability to define which class is "positive" for label encoder in binary classification case #3033
- **Fixes**

  - Fixed bug where `Oversampler` didn't consider boolean columns to be categorical #2980
  - Fixed permutation importance failing when target is categorical #3017
  - Updated estimator and pipelines' `predict`, `predict_proba`, `transform`, `inverse_transform` methods to preserve input indices #2979
  - Updated demo dataset link for daily min temperatures #3023
- **Changes**
  - Updated `OutliersDataCheck` and `UniquenessDataCheck` and allow for the suspension of the Nullable types error #3018
- **Documentation Changes**
  - Fixed cost benefit matrix demo formatting #2990
  - Update ReadMe.md with new badge links and updated installation instructions for conda #2998
  - Added more comprehensive doctests #3002

**v0.36.0 Oct. 27, 2021**

- **Enhancements**
  - Added LIME as an algorithm option for `explain_predictions` and `explain_predictions_best_worst` #2905
  - Standardized data check messages and added default "rows" and "columns" to data check message details dictionary #2869
  - Added `rows_of_interest` to pipeline utils #2908
  - Added support for woodwork version `0.8.2` #2909
  - Enhanced the `DateTimeFeaturizer` to handle `NaNs` in date features #2909
  - Added support for woodwork logical types `PostalCode`, `SubRegionCode`, and `CountryCode` in model understanding tools #2946
  - Added Vowpal Wabbit regressor and classifiers #2846
  - Added *NoSplit* data splitter for future unsupervised learning searches #2958
  - Added method to convert actions into a preprocessing pipeline #2968
- **Fixes**
  - Fixed bug where partial dependence was not respecting the ww schema #2929
  - Fixed `calculate_permutation_importance` for datetimes on `StandardScaler` #2938
  - Fixed `SelectColumns` to only select available features for feature selection in `DefaultAlgorithm` #2944
  - Fixed `DropColumns` component not receiving parameters in `DefaultAlgorithm` #2945
  - Fixed bug where trained binary thresholds were not being returned by `get_pipeline` or `clone` #2948
  - Fixed bug where `Oversampler` selected ww logical categorical instead of ww semantic category #2946
- **Changes**
  - Changed `make_pipeline` function to place the `DateTimeFeaturizer` prior to the `Imputer` so that `NaN` dates can be imputed #2909

  - Refactored `OutliersDataCheck` and `HighlyNullDataCheck` to add more descriptive metadata #2907
  - Bumped minimum version of `dask` from 2021.2.0 to 2021.10.0 #2978
- **Documentation Changes**
  - Added back Future Release section to release notes #2927
  - Updated CI to run doctest (docstring tests) and apply necessary fixes to docstrings #2933
  - Added documentation for `BinaryClassificationPipeline` thresholding #2937
- **Testing Changes**
  - Fixed dependency checker to catch full names of packages #2930
  - Refactored `build_conda_pkg` to work from a local recipe #2925
  - Refactored component test for different environments #2957

**Warning:**

**Breaking Changes**

- Standardized data check messages and added default "rows" and "columns" to data check message details dictionary. This may change the number of messages returned from a data check. #2869

**v0.35.0 Oct. 14, 2021**

- **Enhancements**
  - Added human-readable pipeline explanations to model understanding #2861
  - Updated to support Featuretools 1.0.0 and nlp-primitives 2.0.0 #2848
- **Fixes**
  - Fixed bug where `long` mode for the top level search method was not respected #2875
  - Pinned `cmdstan` to `0.28.0` in `cmdstan-builder` to prevent future breaking of support for Prophet #2880
  - Added `Jarque-Bera` to the `TargetDistributionDataCheck` #2891
- **Changes**
  - Updated pipelines to use a label encoder component instead of doing encoding on the pipeline level #2821
  - Deleted scikit-learn ensembler #2819
  - Refactored pipeline building logic out of `AutoMLSearch` and into `IterativeAlgorithm` #2854
  - Refactored names for methods in `ComponentGraph` and `PipelineBase` #2902
- **Documentation Changes**
  - Updated `install.ipynb` to reflect flexibility for `cmdstan` version installation #2880
  - Updated the conda section of our contributing guide #2899
- **Testing Changes**
  - Updated `test_all_estimators` to account for Prophet being allowed for Python 3.9 #2892
  - Updated linux tests to use `cmdstan-builder==0.0.8` #2880

**Warning:**

**Breaking Changes**

- Updated pipelines to use a label encoder component instead of doing encoding on the pipeline level. This means that pipelines will no longer automatically encode non-numerical targets. Please use a label encoder if working with classification problems and non-numeric targets. #2821
- Deleted scikit-learn ensembler #2819
- `IterativeAlgorithm` now requires X, y, problem_type as required arguments as well as sampler_name, allowed_model_families, allowed_component_graphs, max_batches, and verbose as optional arguments #2854
- Changed method names of `fit_features` and `compute_final_component_features` to `fit_and_transform_all_but_final` and `transform_all_but_final` in `ComponentGraph`, and `compute_estimator_features` to `transform_all_but_final` in pipeline classes #2902

**v0.34.0 Sep. 30, 2021**

- **Enhancements**
  - Updated to work with Woodwork 0.8.1 #2783
  - Added validation that `training_data` and `training_target` are not `None` in prediction explanations #2787
  - Added support for training-only components in pipelines and component graphs #2776
  - Added default argument for the parameters value for `ComponentGraph.instantiate` #2796
  - Added `TIME_SERIES_REGRESSION` to `LightGBMRegressor's` supported problem types #2793
  - Provided a JSON representation of a pipeline's DAG structure #2812
  - Added validation to holdout data passed to `predict` and `predict_proba` for time series #2804
  - Added information about which row indices are outliers in `OutliersDataCheck` #2818
  - Added verbose flag to top level `search()` method #2813
  - Added support for linting jupyter notebooks and clearing the executed cells and empty cells #2829 #2837
  - Added "DROP_ROWS" action to output of `OutliersDataCheck.validate()` #2820
  - Added the ability of `AutoMLSearch` to accept a `SequentialEngine` instance as engine input #2838
  - Added new label encoder component to EvalML #2853
  - Added our own partial dependence implementation #2834
- **Fixes**
  - Fixed bug where `calculate_permutation_importance` was not calculating the right value for pipelines with target transformers #2782
  - Fixed bug where transformed target values were not used in `fit` for time series pipelines #2780
  - Fixed bug where `score_pipelines` method of `AutoMLSearch` would not work for time series problems #2786
  - Removed `TargetTransformer` class #2833
  - Added tests to verify `ComponentGraph` support by pipelines #2830

  - Fixed incorrect parameter for baseline regression pipeline in `AutoMLSearch` #2847
  - Fixed bug where the desired estimator family order was not respected in `IterativeAlgorithm` #2850
- **Changes**
  - Changed woodwork initialization to use partial schemas #2774
  - Made `Transformer.transform()` an abstract method #2744
  - Deleted `EmptyDataChecks` class #2794
  - Removed data check for checking log distributions in `make_pipeline` #2806
  - Changed the minimum `woodwork` version to 0.8.0 #2783
  - Pinned `woodwork` version to 0.8.0 #2832
  - Removed `model_family` attribute from `ComponentBase` and transformers #2828
  - Limited `scikit-learn` until new features and errors can be addressed #2842
  - Show DeprecationWarning when Sklearn Ensemblers are called #2859
- **Testing Changes**
  - Updated matched assertion message regarding monotonic indices in polynomial detrender tests #2811
  - Added a test to make sure pip versions match conda versions #2851

**Warning:**

**Breaking Changes**

- Made `Transformer.transform()` an abstract method #2744
- Deleted `EmptyDataChecks` class #2794
- Removed data check for checking log distributions in `make_pipeline` #2806

**v0.33.0 Sep. 15, 2021**

- **Fixes**
  - Fixed bug where warnings during `make_pipeline` were not being raised to the user #2765
- **Changes**
  - Refactored and removed `SamplerBase` class #2775
- **Documentation Changes**
  - Added docstring linting packages `pydocstyle` and `darglint` to *make-lint* command #2670

**v0.32.1 Sep. 10, 2021**

- **Enhancements**
  - Added `verbose` flag to `AutoMLSearch` to run search in silent mode by default #2645
  - Added label encoder to `XGBoostClassifier` to remove the warning #2701
  - Set `eval_metric` to `logloss` for `XGBoostClassifier` #2741
  - Added support for `woodwork` versions `0.7.0` and `0.7.1` #2743

  - Changed `explain_predictions` functions to display original feature values #2759
  - Added `X_train` and `y_train` to `graph_prediction_vs_actual_over_time` and `get_prediction_vs_actual_over_time_data` #2762
  - Added `forecast_horizon` as a required parameter to time series pipelines and `AutoMLSearch` #2697
  - Added `predict_in_sample` and `predict_proba_in_sample` methods to time series pipelines to predict on data where the target is known, e.g. cross-validation #2697
- **Fixes**
  - Fixed bug where `_catch_warnings` assumed all warnings were `PipelineNotUsed` #2753
  - Fixed bug where `Imputer.transform` would erase ww typing information prior to handing data to the `SimpleImputer` #2752
  - Fixed bug where `Oversampler` could not be copied #2755
- **Changes**
  - Deleted `drop_nan_target_rows` utility method #2737
  - Removed default logging setup and debugging log file #2645
  - Changed the default n_jobs value for `XGBoostClassifier` and `XGBoostRegressor` to 12 #2757
  - Changed `TimeSeriesBaselineEstimator` to only work on a time series pipeline with a `DelayedFeaturesTransformer` #2697
  - Added `X_train` and `y_train` as optional parameters to pipeline `predict`, `predict_proba`. Only used for time series pipelines #2697
  - Added `training_data` and `training_target` as optional parameters to `explain_predictions` and `explain_predictions_best_worst` to support time series pipelines #2697
  - Changed time series pipeline predictions to no longer output series/dataframes padded with NaNs. A prediction will be returned for every row in the *X* input #2697
- **Documentation Changes**
  - Specified installation steps for Prophet #2713
  - Added documentation for data exploration on data check actions #2696
  - Added a user guide entry for time series modelling #2697
- **Testing Changes**
  - Fixed flaky `TargetDistributionDataCheck` test for very_lognormal distribution #2748

**Warning:**

**Breaking Changes**

- Removed default logging setup and debugging log file #2645
- Added `X_train` and `y_train` to `graph_prediction_vs_actual_over_time` and `get_prediction_vs_actual_over_time_data` #2762
- Added `forecast_horizon` as a required parameter to time series pipelines and `AutoMLSearch` #2697
- Changed `TimeSeriesBaselineEstimator` to only work on a time series pipeline with a `DelayedFeaturesTransformer` #2697

- Added `X_train` and `y_train` as required parameters for `predict` and `predict_proba` in time series pipelines #2697
- Added `training_data` and `training_target` as required parameters to `explain_predictions` and `explain_predictions_best_worst` for time series pipelines #2697

**v0.32.0 Aug. 31, 2021**

- **Enhancements**
  - Allow string for `engine` parameter for `AutoMLSearch`#2667
  - Add `ProphetRegressor` to AutoML #2619
  - Integrated `DefaultAlgorithm` into `AutoMLSearch` #2634
  - Removed SVM "linear" and "precomputed" kernel hyperparameter options, and improved default parameters #2651
  - Updated `ComponentGraph` initalization to raise `ValueError` when user attempts to use `.y` for a component that does not produce a tuple output #2662
  - Updated to support Woodwork 0.6.0 #2690
  - Updated pipeline `graph()` to distingush X and y edges #2654
  - Added `DropRowsTransformer` component #2692
  - Added `DROP_ROWS` to `_make_component_list_from_actions` and clean up metadata #2694
  - Add new ensembler component #2653
- **Fixes**
  - Updated Oversampler logic to select best SMOTE based on component input instead of pipeline input #2695
  - Added ability to explicitly close DaskEngine resources to improve runtime and reduce Dask warnings #2667
  - Fixed partial dependence bug for ensemble pipelines #2714
  - Updated `TargetLeakageDataCheck` to maintain user-selected logical types #2711
- **Changes**
  - Replaced `SMOTEOversampler`, `SMOTENOversampler` and `SMOTENCOversampler` with consolidated `Oversampler` component #2695
  - Removed `LinearRegressor` from the list of default `AutoMLSearch` estimators due to poor performance #2660
- **Documentation Changes**
  - Added user guide documentation for using `ComponentGraph` and added `ComponentGraph` to API reference #2673
  - Updated documentation to make parallelization of AutoML clearer #2667
- **Testing Changes**
  - Removes the process-level parallelism from the `test_cancel_job` test #2666
  - Installed numba 0.53 in windows CI to prevent problems installing version 0.54 #2710

**Warning:**

**Breaking Changes**

- Renamed the current top level `search` method to `search_iterative` and defined a new `search` method for the `DefaultAlgorithm` #2634
- Replaced `SMOTEOversampler`, `SMOTENOversampler` and `SMOTENCOversampler` with consolidated `Oversampler` component #2695
- Removed `LinearRegressor` from the list of default `AutoMLSearch` estimators due to poor performance #2660

**v0.31.0 Aug. 19, 2021**

- **Enhancements**
  - Updated the high variance check in AutoMLSearch to be robust to a variety of objectives and cv scores #2622
  - Use Woodwork's outlier detection for the `OutliersDataCheck` #2637
  - Added ability to utilize instantiated components when creating a pipeline #2643
  - Sped up the all Nan and unknown check in `infer_feature_types` #2661
- Fixes
- **Changes**
  - Deleted `_put_into_original_order` helper function #2639
  - Refactored time series pipeline code using a time series pipeline base class #2649
  - Renamed `dask_tests` to `parallel_tests` #2657
  - Removed commented out code in `pipeline_meta.py` #2659
- **Documentation Changes**
  - Add complete install command to README and Install section #2627
  - Cleaned up documentation for `MulticollinearityDataCheck` #2664
- **Testing Changes**
  - Speed up CI by splitting Prophet tests into a separate workflow in GitHub #2644

**Warning:**

**Breaking Changes**

- `TimeSeriesRegressionPipeline` no longer inherits from `TimeSeriesRegressionPipeline` #2649

**v0.30.2 Aug. 16, 2021**

- **Fixes**
  - Updated changelog and version numbers to match the release. Release 0.30.1 was release erroneously without a change to the version numbers. 0.30.2 replaces it.

**v0.30.1 Aug. 12, 2021**

- **Enhancements**

  - Added `DatetimeFormatDataCheck` for time series problems #2603
  - Added `ProphetRegressor` to estimators #2242
  - Updated `ComponentGraph` to handle not calling samplers' transform during predict, and updated samplers' transform methods s.t. `fit_transform` is equivalent to `fit(X, y).transform(X, y)` #2583
  - Updated `ComponentGraph` `_validate_component_dict` logic to be stricter about input values #2599
  - Patched bug in `xgboost` estimators where predicting on a feature matrix of only booleans would throw an exception. #2602
  - Updated `ARIMARegressor` to use relative forecasting to predict values #2613
  - Added support for creating pipelines without an estimator as the final component and added `transform(X, y)` method to pipelines and component graphs #2625
  - Updated to support Woodwork 0.5.1 #2610
- **Fixes**
  - Updated `AutoMLSearch` to drop `ARIMARegressor` from `allowed_estimators` if an incompatible frequency is detected #2632
  - Updated `get_best_sampler_for_data` to consider all non-numeric datatypes as categorical for SMOTE #2590
  - Fixed inconsistent test results from *TargetDistributionDataCheck* #2608
  - Adopted vectorized pd.NA checking for Woodwork 0.5.1 support #2626
  - Pinned upper version of astroid to 2.6.6 to keep ReadTheDocs working. #2638
- **Changes**
  - Renamed SMOTE samplers to SMOTE oversampler #2595
  - Changed `partial_dependence` and `graph_partial_dependence` to raise a `PartialDependenceError` instead of `ValueError`. This is not a breaking change because `PartialDependenceError` is a subclass of `ValueError` #2604
  - Cleaned up code duplication in `ComponentGraph` #2612
  - Stored predict_proba results in .x for intermediate estimators in ComponentGraph #2629
- **Documentation Changes**
  - To avoid local docs build error, only add warning disable and download headers on ReadTheDocs builds, not locally #2617
- **Testing Changes**
  - Updated partial_dependence tests to change the element-wise comparison per the Plotly 5.2.1 upgrade #2638
  - Changed the lint CI job to only check against python 3.9 via the *-t* flag #2586
  - Installed Prophet in linux nightlies test and fixed `test_all_components` #2598
  - Refactored and fixed all `make_pipeline` tests to assert correct order and address new Woodwork Unknown type inference #2572
  - Removed `component_graphs` as a global variable in `test_component_graphs.py` #2609

**Warning:**

**Breaking Changes**

- Renamed SMOTE samplers to SMOTE oversampler. Please use `SMOTEOversampler`, `SMOTENCOversampler`, `SMOTENOversampler` instead of `SMOTESampler`, `SMOTENCSampler`, and `SMOTENSampler` #2595

**v0.30.0 Aug. 3, 2021**

- **Enhancements**
  - Added `LogTransformer` and `TargetDistributionDataCheck` #2487
  - Issue a warning to users when a pipeline parameter passed in isn't used in the pipeline #2564
  - Added Gini coefficient as an objective #2544
  - Added `repr` to `ComponentGraph` #2565
  - Added components to extract features from `URL` and `EmailAddress` Logical Types #2550
  - Added support for *NaN* values in `TextFeaturizer` #2532
  - Added `SelectByType` transformer #2531
  - Added separate thresholds for percent null rows and columns in `HighlyNullDataCheck` #2562
  - Added support for *NaN* natural language values #2577
- **Fixes**
  - Raised error message for types `URL`, `NaturalLanguage`, and `EmailAddress` in `partial_dependence` #2573
- **Changes**
  - Updated `PipelineBase` implementation for creating pipelines from a list of components #2549
  - Moved `get_hyperparameter_ranges` to `PipelineBase` class from automl/utils module #2546
  - Renamed `ComponentGraph`'s `get_parents` to `get_inputs` #2540
  - Removed `ComponentGraph.linearized_component_graph` and `ComponentGraph.from_list` #2556
  - Updated `ComponentGraph` to enforce requiring *.x* and *.y* inputs for each component in the graph #2563
  - Renamed existing ensembler implementation from `StackedEnsemblers` to `SklearnStackedEnsemblers` #2578
- **Documentation Changes**
  - Added documentation for `DaskEngine` and `CFEngine` parallel engines #2560
  - Improved detail of `TextFeaturizer` docstring and tutorial #2568
- **Testing Changes**
  - Added test that makes sure `split_data` does not shuffle for time series problems #2552

**Warning:**

**Breaking Changes**

- Moved `get_hyperparameter_ranges` to `PipelineBase` class from automl/utils module #2546
- Renamed `ComponentGraph`'s `get_parents` to `get_inputs` #2540
- Removed `ComponentGraph.linearized_component_graph` and `ComponentGraph.from_list` #2556
- Updated `ComponentGraph` to enforce requiring *.x* and *.y* inputs for each component in the graph #2563

**v0.29.0 Jul. 21, 2021**

- **Enhancements**
  - Updated 1-way partial dependence support for datetime features #2454
  - Added details on how to fix error caused by broken ww schema #2466
  - Added ability to use built-in pickle for saving AutoMLSearch #2463
  - Updated our components and component graphs to use latest features of ww 0.4.1, e.g. `concat_columns` and drop in-place. #2465
  - Added new, concurrent.futures based engine for parallel AutoML #2506
  - Added support for new Woodwork `Unknown` type in AutoMLSearch #2477
  - Updated our components with an attribute that describes if they modify features or targets and can be used in list API for pipeline initialization #2504
  - Updated `ComponentGraph` to accept X and y as inputs #2507
  - Removed unused `TARGET_BINARY_INVALID_VALUES` from `DataCheckMessageCode` enum and fixed formatting of objective documentation #2520
  - Added `EvalMLAlgorithm` #2525
  - Added support for *NaN* values in `TextFeaturizer` #2532
- **Fixes**
  - Fixed `FraudCost` objective and reverted threshold optimization method for binary classification to `Golden` #2450
  - Added custom exception message for partial dependence on features with scales that are too small #2455
  - Ensures the typing for Ordinal and Datetime ltypes are passed through _retain_custom_types_and_initalize_woodwork #2461
  - Updated to work with Pandas 1.3.0 #2442
  - Updated to work with sktime 0.7.0 #2499
- **Changes**
  - Updated XGBoost dependency to >=`1.4.2` #2484, #2498
  - Added a `DeprecationWarning` about deprecating the list API for `ComponentGraph` #2488
  - Updated `make_pipeline` for AutoML to create dictionaries, not lists, to initialize pipelines #2504
  - No longer installing graphviz on windows in our CI pipelines because release 0.17 breaks windows 3.7 #2516
- **Documentation Changes**

  - Moved docstrings from `__init__` to class pages, added missing docstrings for missing classes, and updated missing default values #2452
  - Build documentation with sphinx-autoapi #2458
  - Change `autoapi_ignore` to only ignore files in `evalml/tests/*` #2530
- **Testing Changes**
  - Fixed flaky dask tests #2471
  - Removed shellcheck action from `build_conda_pkg` action #2514
  - Added a tmp_dir fixture that deletes its contents after tests run #2505
  - Added a test that makes sure all pipelines in `AutoMLSearch` get the same data splits #2513
  - Condensed warning output in test logs #2521

**Warning:**

**Breaking Changes**

- *NaN* values in the *Natural Language* type are no longer supported by the Imputer with the pandas upgrade. #2477

**v0.28.0 Jul. 2, 2021**

- **Enhancements**
  - Added support for showing a Individual Conditional Expectations plot when graphing Partial Dependence #2386
  - Exposed `thread_count` for Catboost estimators as `n_jobs` parameter #2410
  - Updated Objectives API to allow for sample weighting #2433
- **Fixes**
  - Deleted unreachable line from `IterativeAlgorithm` #2464
- **Changes**
  - Pinned Woodwork version between 0.4.1 and 0.4.2 #2460
  - Updated psutils minimum version in requirements #2438
  - Updated `log_error_callback` to not include filepath in logged message #2429
- **Documentation Changes**
  - Sped up docs #2430
  - Removed mentions of `DataTable` and `DataColumn` from the docs #2445
- **Testing Changes**
  - Added slack integration for nightlies tests #2436
  - Changed `build_conda_pkg` CI job to run only when dependencies are updates #2446
  - Updated workflows to store pytest runtimes as test artifacts #2448
  - Added `AutoMLTestEnv` test fixture for making it easy to mock automl tests #2406

**v0.27.0 Jun. 22, 2021**

- **Enhancements**

  - Adds force plots for prediction explanations #2157
  - Removed self-reference from `AutoMLSearch` #2304
  - Added support for nonlinear pipelines for `generate_pipeline_code` #2332
  - Added `inverse_transform` method to pipelines #2256
  - Add optional automatic update checker #2350
  - Added `search_order` to `AutoMLSearch`'s `rankings` and `full_rankings` tables #2345
  - Updated threshold optimization method for binary classification #2315
  - Updated demos to pull data from S3 instead of including demo data in package #2387
  - Upgrade woodwork version to v0.4.1 #2379
- **Fixes**
  - Preserve user-specified woodwork types throughout pipeline fit/predict #2297
  - Fixed `ComponentGraph` appending target to `final_component_features` if there is a component that returns both X and y #2358
  - Fixed partial dependence graph method failing on multiclass problems when the class labels are numeric #2372
  - Added `thresholding_objective` argument to `AutoMLSearch` for binary classification problems #2320
  - Added change for `k_neighbors` parameter in SMOTE Oversamplers to automatically handle small samples #2375
  - Changed naming for `Logistic Regression Classifier` file #2399
  - Pinned pytest-timeout to fix minimum dependence checker #2425
  - Replaced `Elastic Net Classifier` base class with `Logistsic Regression` to avoid `NaN` outputs #2420
- **Changes**
  - Cleaned up `PipelineBase`'s `component_graph` and `_component_graph` attributes. Updated `PipelineBase` `__repr__` and added `__eq__` for `ComponentGraph` #2332
  - Added and applied `black` linting package to the EvalML repo in place of `autopep8` #2306
  - Separated *custom_hyperparameters* from pipelines and added them as an argument to `AutoMLSearch` #2317
  - Replaced *allowed_pipelines* with *allowed_component_graphs* #2364
  - Removed private method `_compute_features_during_fit` from `PipelineBase` #2359
  - Updated `compute_order` in `ComponentGraph` to be a read-only property #2408
  - Unpinned PyZMQ version in requirements.txt #2389
  - Uncapping LightGBM version in requirements.txt #2405
  - Updated minimum version of plotly #2415
  - Removed `SensitivityLowAlert` objective from core objectives #2418
- **Documentation Changes**
  - Fixed lead scoring weights in the demos documentation #2315

  - Fixed start page code and description dataset naming discrepancy #2370
- **Testing Changes**
  - Update minimum unit tests to run on all pull requests #2314
  - Pass token to authorize uploading of codecov reports #2344
  - Add `pytest-timeout`. All tests that run longer than 6 minutes will fail. #2374
  - Separated the dask tests out into separate github action jobs to isolate dask failures. #2376
  - Refactored dask tests #2377
  - Added the combined dask/non-dask unit tests back and renamed the dask only unit tests. #2382
  - Sped up unit tests and split into separate jobs #2365
  - Change CI job names, run lint for python 3.9, run nightlies on python 3.8 at 3am EST #2395 #2398
  - Set fail-fast to false for CI jobs that run for PRs #2402

**Warning:**

**Breaking Changes**

- *AutoMLSearch* will accept *allowed_component_graphs* instead of *allowed_pipelines* #2364
- Removed `PipelineBase`'s `_component_graph` attribute. Updated `PipelineBase __repr__` and added `__eq__` for `ComponentGraph` #2332
- *pipeline_parameters* will no longer accept *skopt.space* variables since hyperparameter ranges will now be specified through *custom_hyperparameters* #2317

**v0.25.0 Jun. 01, 2021**

- **Enhancements**
  - Upgraded minimum woodwork to version 0.3.1. Previous versions will not be supported #2181
  - Added a new callback parameter for `explain_predictions_best_worst` #2308
- Fixes
- **Changes**
  - Deleted the `return_pandas` flag from our demo data loaders #2181
  - Moved `default_parameters` to `ComponentGraph` from `PipelineBase` #2307
- **Documentation Changes**
  - Updated the release procedure documentation #2230
- **Testing Changes**
  - Ignoring `test_saving_png_file` while building conda package #2323

**Warning:**

**Breaking Changes**

- Deleted the `return_pandas` flag from our demo data loaders #2181
- Upgraded minimum woodwork to version 0.3.1. Previous versions will not be supported #2181

- Due to the weak-ref in woodwork, set the result of `infer_feature_types` to a variable before accessing woodwork #2181

**v0.24.2 May. 24, 2021**

- **Enhancements**
  - Added oversamplers to AutoMLSearch #2213 #2286
  - Added dictionary input functionality for `Undersampler` component #2271
  - Changed the default parameter values for `Elastic Net Classifier` and `Elastic Net Regressor` #2269
  - Added dictionary input functionality for the Oversampler components #2288
- **Fixes**
  - Set default *n_jobs* to 1 for *StackedEnsembleClassifier* and *StackedEnsembleRegressor* until fix for text-based parallelism in sklearn stacking can be found #2295
- **Changes**
  - Updated `start_iteration_callback` to accept a pipeline instance instead of a pipeline class and no longer accept pipeline parameters as a parameter #2290
  - Refactored `calculate_permutation_importance` method and add per-column permutation importance method #2302
  - Updated logging information in `AutoMLSearch.__init__` to clarify pipeline generation #2263
- **Documentation Changes**
  - Minor changes to the release procedure #2230
- **Testing Changes**
  - Use codecov action to update coverage reports #2238
  - Removed MarkupSafe dependency version pin from requirements.txt and moved instead into RTD docs build CI #2261

**Warning:**

**Breaking Changes**

- Updated `start_iteration_callback` to accept a pipeline instance instead of a pipeline class and no longer accept pipeline parameters as a parameter #2290
- Moved `default_parameters` to `ComponentGraph` from `PipelineBase`. A pipeline's `default_parameters` is now accessible via `pipeline.component_graph.default_parameters` #2307

**v0.24.1 May. 16, 2021**

- **Enhancements**
  - Integrated `ARIMARegressor` into AutoML #2009
  - Updated `HighlyNullDataCheck` to also perform a null row check #2222
  - Set `max_depth` to 1 in calls to featuretools dfs #2231
- **Fixes**

  - Removed data splitter sampler calls during training #2253
  - Set minimum required version for for pyzmq, colorama, and docutils #2254
  - Changed BaseSampler to return None instead of y #2272
- **Changes**
  - Removed ensemble split and indices in `AutoMLSearch` #2260
  - Updated pipeline `repr()` and `generate_pipeline_code` to return pipeline instances without generating custom pipeline class #2227
- **Documentation Changes**
  - Capped Sphinx version under 4.0.0 #2244
- **Testing Changes**
  - Change number of cores for pytest from 4 to 2 #2266
  - Add minimum dependency checker to generate minimum requirement files #2267
  - Add unit tests with minimum dependencies #2277

**v0.24.0 May. 04, 2021**

- **Enhancements**
  - Added *date_index* as a required parameter for TimeSeries problems #2217
  - Have the `OneHotEncoder` return the transformed columns as booleans rather than floats #2170
  - Added Oversampler transformer component to EvalML #2079
  - Added Undersampler to AutoMLSearch, as well as arguments `_sampler_method` and `sampler_balanced_ratio` #2128
  - Updated prediction explanations functions to allow pipelines with XGBoost estimators #2162
  - Added partial dependence for datetime columns #2180
  - Update precision-recall curve with positive label index argument, and fix for 2d predicted probabilities #2090
  - Add pct_null_rows to `HighlyNullDataCheck` #2211
  - Added a standalone AutoML *search* method for convenience, which runs data checks and then runs automl #2152
  - Make the first batch of AutoML have a predefined order, with linear models first and complex models last #2223 #2225
  - Added sampling dictionary support to `BalancedClassficationSampler` #2235
- **Fixes**
  - Fixed partial dependence not respecting grid resolution parameter for numerical features #2180
  - Enable prediction explanations for catboost for multiclass problems #2224
- **Changes**
  - Deleted baseline pipeline classes #2202
  - Reverting user specified date feature PR #2155 until *pmdarima* installation fix is found #2214
  - Updated pipeline API to accept component graph and other class attributes as instance parameters. Old pipeline API still works but will not be supported long-term. #2091

  - Removed all old datasplitters from EvalML #2193
  - Deleted `make_pipeline_from_components` #2218
- **Documentation Changes**
  - Renamed dataset to clarify that its gzipped but not a tarball #2183
  - Updated documentation to use pipeline instances instead of pipeline subclasses #2195
  - Updated contributing guide with a note about GitHub Actions permissions #2090
  - Updated automl and model understanding user guides #2090
- **Testing Changes**
  - Use machineFL user token for dependency update bot, and add more reviewers #2189

**Warning:**

**Breaking Changes**

- All baseline pipeline classes (`BaselineBinaryPipeline`, `BaselineMulticlassPipeline`, `BaselineRegressionPipeline`, etc.) have been deleted #2202
- Updated pipeline API to accept component graph and other class attributes as instance parameters. Old pipeline API still works but will not be supported long-term. Pipelines can now be initialized by specifying the component graph as the first parameter, and then passing in optional arguments such as `custom_name`, `parameters`, etc. For example, `BinaryClassificationPipeline(["Random Forest Classifier"], parameters={})`. #2091
- Removed all old datasplitters from EvalML #2193
- Deleted utility method `make_pipeline_from_components` #2218

**v0.23.0 Apr. 20, 2021**

- **Enhancements**
  - Refactored `EngineBase` and `SequentialEngine` api. Adding `DaskEngine` #1975.
  - Added optional `engine` argument to `AutoMLSearch` #1975
  - Added a warning about how time series support is still in beta when a user passes in a time series problem to `AutoMLSearch` #2118
  - Added `NaturalLanguageNaNDataCheck` data check #2122
  - Added ValueError to `partial_dependence` to prevent users from computing partial dependence on columns with all NaNs #2120
  - Added standard deviation of cv scores to rankings table #2154
- **Fixes**
  - Fixed `BalancedClassificationDataCVSplit`, `BalancedClassificationDataTVSplit`, and `BalancedClassificationSampler` to use `minority:majority` ratio instead of `majority:minority` #2077
  - Fixed bug where two-way partial dependence plots with categorical variables were not working correctly #2117
  - Fixed bug where `hyperparameters` were not displaying properly for pipelines with a list `component_graph` and duplicate components #2133

  - Fixed bug where `pipeline_parameters` argument in `AutoMLSearch` was not applied to pipelines passed in as `allowed_pipelines` #2133
  - Fixed bug where `AutoMLSearch` was not applying custom hyperparameters to pipelines with a list `component_graph` and duplicate components #2133
- **Changes**
  - Removed `hyperparameter_ranges` from Undersampler and renamed `balanced_ratio` to `sampling_ratio` for samplers #2113
  - Renamed `TARGET_BINARY_NOT_TWO_EXAMPLES_PER_CLASS` data check message code to `TARGET_MULTICLASS_NOT_TWO_EXAMPLES_PER_CLASS` #2126
  - Modified one-way partial dependence plots of categorical features to display data with a bar plot #2117
  - Renamed `score` column for `automl.rankings` as `mean_cv_score` #2135
  - Remove 'warning' from docs tool output #2031
- **Documentation Changes**
  - Fixed `conf.py` file #2112
  - Added a sentence to the automl user guide stating that our support for time series problems is still in beta. #2118
  - Fixed documentation demos #2139
  - Update test badge in README to use GitHub Actions #2150
- **Testing Changes**
  - Fixed `test_describe_pipeline` for `pandas v1.2.4` #2129
  - Added a GitHub Action for building the conda package #1870 #2148

**Warning:**

**Breaking Changes**

- Renamed `balanced_ratio` to `sampling_ratio` for the `BalancedClassificationDataCVSplit`, `BalancedClassificationDataTVSplit`, `BalancedClassficationSampler`, and Undersampler #2113
- Deleted the "errors" key from automl results #1975
- Deleted the `raise_and_save_error_callback` and the `log_and_save_error_callback` #1975
- Fixed `BalancedClassificationDataCVSplit`, `BalancedClassificationDataTVSplit`, and `BalancedClassificationSampler` to use minority:majority ratio instead of majority:minority #2077

**v0.22.0 Apr. 06, 2021**

- **Enhancements**
  - Added a GitHub Action for `linux_unit_tests`#2013
  - Added recommended actions for `InvalidTargetDataCheck`, updated `_make_component_list_from_actions` to address new action, and added `TargetImputer` component #1989
  - Updated `AutoMLSearch._check_for_high_variance` to not emit `RuntimeWarning` #2024

  - Added exception when pipeline passed to `explain_predictions` is a `Stacked Ensemble` pipeline #2033
  - Added sensitivity at low alert rates as an objective #2001
  - Added `Undersampler` transformer component #2030
- **Fixes**
  - Updated Engine's `train_batch` to apply undersampling #2038
  - Fixed bug in where Time Series Classification pipelines were not encoding targets in `predict` and `predict_proba` #2040
  - Fixed data splitting errors if target is float for classification problems #2050
  - Pinned `docutils` to <0.17 to fix ReadtheDocs warning issues #2088
- **Changes**
  - Removed lists as acceptable hyperparameter ranges in `AutoMLSearch` #2028
  - Renamed "details" to "metadata" for data check actions #2008
- **Documentation Changes**
  - Catch and suppress warnings in documentation #1991 #2097
  - Change spacing in `start.ipynb` to provide clarity for `AutoMLSearch` #2078
  - Fixed start code on README #2108
- Testing Changes

**v0.21.0 Mar. 24, 2021**

- **Enhancements**
  - Changed `AutoMLSearch` to default `optimize_thresholds` to True #1943
  - Added multiple oversampling and undersampling sampling methods as data splitters for imbalanced classification #1775
  - Added params to balanced classification data splitters for visibility #1966
  - Updated `make_pipeline` to not add `Imputer` if input data does not have numeric or categorical columns #1967
  - Updated `ClassImbalanceDataCheck` to better handle multiclass imbalances #1986
  - Added recommended actions for the output of data check's `validate` method #1968
  - Added error message for `partial_dependence` when features are mostly the same value #1994
  - Updated `OneHotEncoder` to drop one redundant feature by default for features with two categories #1997
  - Added a `PolynomialDecomposer` component #1992
  - Added `DateTimeNaNDataCheck` data check #2039
- **Fixes**
  - Changed best pipeline to train on the entire dataset rather than just ensemble indices for ensemble problems #2037
  - Updated binary classification pipelines to use objective decision function during scoring of custom objectives #1934

- **Changes**
  - Removed `data_checks` parameter, `data_check_results` and data checks logic from `AutoMLSearch` #1935
  - Deleted `random_state` argument #1985
  - Updated Woodwork version requirement to `v0.0.11` #1996
- Documentation Changes
- **Testing Changes**
  - Removed `build_docs` CI job in favor of RTD GH builder #1974
  - Added tests to confirm support for Python 3.9 #1724
  - Added tests to support Dask AutoML/Engine #1990
  - Changed `build_conda_pkg` job to use `latest_release_changes` branch in the feedstock. #1979

**Warning:**

**Breaking Changes**

- Changed `AutoMLSearch` to default `optimize_thresholds` to True #1943
- Removed `data_checks` parameter, `data_check_results` and data checks logic from `AutoMLSearch`. To run the data checks which were previously run by default in `AutoMLSearch`, please call `DefaultDataChecks().validate(X_train, y_train)` or take a look at our documentation for more examples. #1935
- Deleted `random_state` argument #1985

**v0.20.0 Mar. 10, 2021**

- **Enhancements**
  - Added a GitHub Action for Detecting dependency changes #1933
  - Create a separate CV split to train stacked ensembler on for AutoMLSearch #1814
  - Added a GitHub Action for Linux unit tests #1846
  - Added `ARIMARegressor` estimator #1894
  - Added `DataCheckAction` class and `DataCheckActionCode` enum #1896
  - Updated `Woodwork` requirement to `v0.0.10` #1900
  - Added `BalancedClassificationDataCVSplit` and `BalancedClassificationDataTVSplit` to AutoMLSearch #1875
  - Update default classification data splitter to use downsampling for highly imbalanced data #1875
  - Updated `describe_pipeline` to return more information, including `id` of pipelines used for ensemble models #1909
  - Added utility method to create list of components from a list of `DataCheckAction` #1907
  - Updated `validate` method to include a `action` key in returned dictionary for all `DataCheck``and ``DataChecks` #1916
  - Aggregating the shap values for predictions that we know the provenance of, e.g. OHE, text, and date-time. #1901

  - Improved error message when custom objective is passed as a string in `pipeline.score` #1941
  - Added `score_pipelines` and `train_pipelines` methods to `AutoMLSearch` #1913
  - Added support for `pandas` version 1.2.0 #1708
  - Added `score_batch` and `train_batch` abstact methods to `EngineBase` and implementations in `SequentialEngine` #1913
  - Added ability to handle index columns in `AutoMLSearch` and `DataChecks` #2138
- **Fixes**
  - Removed CI check for `check_dependencies_updated_linux` #1950
  - Added metaclass for time series pipelines and fix binary classification pipeline `predict` not using objective if it is passed as a named argument #1874
  - Fixed stack trace in prediction explanation functions caused by mixed string/numeric pandas column names #1871
  - Fixed stack trace caused by passing pipelines with duplicate names to `AutoMLSearch` #1932
  - Fixed `AutoMLSearch.get_pipelines` returning pipelines with the same attributes #1958
- **Changes**
  - Reversed GitHub Action for Linux unit tests until a fix for report generation is found #1920
  - Updated `add_results` in `AutoMLAlgorithm` to take in entire pipeline results dictionary from `AutoMLSearch` #1891
  - Updated `ClassImbalanceDataCheck` to look for severe class imbalance scenarios #1905
  - Deleted the `explain_prediction` function #1915
  - Removed `HighVarianceCVDataCheck` and convered it to an `AutoMLSearch` method instead #1928
  - Removed warning in `InvalidTargetDataCheck` returned when numeric binary classification targets are not (0, 1) #1959
- **Documentation Changes**
  - Updated `model_understanding.ipynb` to demo the two-way partial dependence capability #1919
- Testing Changes

**Warning:**

**Breaking Changes**

- Deleted the `explain_prediction` function #1915
- Removed `HighVarianceCVDataCheck` and convered it to an `AutoMLSearch` method instead #1928
- Added `score_batch` and `train_batch` abstact methods to `EngineBase`. These need to be implemented in Engine subclasses #1913

**v0.19.0 Feb. 23, 2021**

- **Enhancements**
  - Added a GitHub Action for Python windows unit tests #1844

  - Added a GitHub Action for checking updated release notes #1849
  - Added a GitHub Action for Python lint checks #1837
  - Adjusted `explain_prediction`, `explain_predictions` and `explain_predictions_best_worst` to handle timeseries problems. #1818
  - Updated `InvalidTargetDataCheck` to check for mismatched indices in target and features #1816
  - Updated `Woodwork` structures returned from components to support `Woodwork` logical type overrides set by the user #1784
  - Updated estimators to keep track of input feature names during `fit()` #1794
  - Updated `visualize_decision_tree` to include feature names in output #1813
  - Added `is_bounded_like_percentage` property for objectives. If true, the `calculate_percent_difference` method will return the absolute difference rather than relative difference #1809
  - Added full error traceback to AutoMLSearch logger file #1840
  - Changed `TargetEncoder` to preserve custom indices in the data #1836
  - Refactored `explain_predictions` and `explain_predictions_best_worst` to only compute features once for all rows that need to be explained #1843
  - Added custom random undersampler data splitter for classification #1857
  - Updated `OutliersDataCheck` implementation to calculate the probability of having no outliers #1855
  - Added `Engines` pipeline processing API #1838
- **Fixes**
  - Changed EngineBase random_state arg to random_seed and same for user guide docs #1889
- **Changes**
  - Modified `calculate_percent_difference` so that division by 0 is now inf rather than nan #1809
  - Removed `text_columns` parameter from `LSA` and `TextFeaturizer` components #1652
  - Added `random_seed` as an argument to our automl/pipeline/component API. Using `random_state` will raise a warning #1798
  - Added `DataCheckError` message in `InvalidTargetDataCheck` if input target is None and removed exception raised #1866
- Documentation Changes
- **Testing Changes**
  - Added back coverage for `_get_feature_provenance` in `TextFeaturizer` after `text_columns` was removed #1842
  - Pin graphviz version for windows builds #1847
  - Unpin graphviz version for windows builds #1851

**Warning:**

**Breaking Changes**

- Added a deprecation warning to `explain_prediction`. It will be deleted in the next release. #1860

**v0.18.2 Feb. 10, 2021**

- **Enhancements**
  - Added uniqueness score data check #1785
  - Added "dataframe" output format for prediction explanations #1781
  - Updated LightGBM estimators to handle `pandas.MultiIndex` #1770
  - Sped up permutation importance for some pipelines #1762
  - Added sparsity data check #1797
  - Confirmed support for threshold tuning for binary time series classification problems #1803
- Fixes
- Changes
- **Documentation Changes**
  - Added section on conda to the contributing guide #1771
  - Updated release process to reflect freezing *main* before perf tests #1787
  - Moving some prs to the right section of the release notes #1789
  - Tweak README.md. #1800
  - Fixed back arrow on install page docs #1795
  - Fixed docstring for *ClassImbalanceDataCheck.validate()* #1817
- Testing Changes

**v0.18.1 Feb. 1, 2021**

- **Enhancements**
  - Added `graph_t_sne` as a visualization tool for high dimensional data #1731
  - Added the ability to see the linear coefficients of features in linear models terms #1738
  - Added support for `scikit-learn` `v0.24.0` #1733
  - Added support for `scipy` `v1.6.0` #1752
  - Added SVM Classifier and Regressor to estimators #1714 #1761
- **Fixes**
  - Addressed bug with `partial_dependence` and categorical data with more categories than grid resolution #1748
  - Removed `random_state` arg from `get_pipelines` in `AutoMLSearch` #1719
  - Pinned pyzmq at less than 22.0.0 till we add support #1756
- **Changes**
  - Updated components and pipelines to return `Woodwork` data structures #1668
  - Updated `clone()` for pipelines and components to copy over random state automatically #1753
  - Dropped support for Python version 3.6 #1751

  - Removed deprecated `verbose` flag from `AutoMLSearch` parameters #1772
- **Documentation Changes**
  - Add Twitter and Github link to documentation toolbar #1754
  - Added Open Graph info to documentation #1758
- Testing Changes

**Warning:**

**Breaking Changes**

- Components and pipelines return `Woodwork` data structures instead of `pandas` data structures #1668
- Python 3.6 will not be actively supported due to discontinued support from EvalML dependencies.
- Deprecated `verbose` flag is removed for `AutoMLSearch` #1772

**v0.18.0 Jan. 26, 2021**

- **Enhancements**
  - Added RMSLE, MSLE, and MAPE to core objectives while checking for negative target values in `invalid_targets_data_check` #1574
  - Added validation checks for binary problems with regression-like datasets and multiclass problems without true multiclass targets in `invalid_targets_data_check` #1665
  - Added time series support for `make_pipeline` #1566
  - Added target name for output of pipeline `predict` method #1578
  - Added multiclass check to `InvalidTargetDataCheck` for two examples per class #1596
  - Added support for `graphviz v0.16` #1657
  - Enhanced time series pipelines to accept empty features #1651
  - Added KNN Classifier to estimators. #1650
  - Added support for list inputs for objectives #1663
  - Added support for `AutoMLSearch` to handle time series classification pipelines #1666
  - Enhanced `DelayedFeaturesTransformer` to encode categorical features and targets before delaying them #1691
  - Added 2-way dependence plots. #1690
  - Added ability to directly iterate through components within Pipelines #1583
- **Fixes**
  - Fixed inconsistent attributes and added Exceptions to docs #1673
  - Fixed `TargetLeakageDataCheck` to use Woodwork `mutual_information` rather than using Pandas' Pearson Correlation #1616
  - Fixed thresholding for pipelines in `AutoMLSearch` to only threshold binary classification pipelines #1622 #1626
  - Updated `load_data` to return Woodwork structures and update default parameter value for `index` to `None` #1610
  - Pinned scipy at < 1.6.0 while we work on adding support #1629

  - Fixed data check message formatting in `AutoMLSearch` #1633
  - Addressed stacked ensemble component for `scikit-learn` v0.24 support by setting `shuffle=True` for default CV #1613
  - Fixed bug where `Imputer` reset the index on `X` #1590
  - Fixed `AutoMLSearch` stacktrace when a cutom objective was passed in as a primary objective or additional objective #1575
  - Fixed custom index bug for `MAPE` objective #1641
  - Fixed index bug for `TextFeaturizer` and `LSA` components #1644
  - Limited `load_fraud` dataset loaded into `automl.ipynb` #1646
  - `add_to_rankings` updates `AutoMLSearch.best_pipeline` when necessary #1647
  - Fixed bug where time series baseline estimators were not receiving `gap` and `max_delay` in `AutoMLSearch` #1645
  - Fixed jupyter notebooks to help the RTD buildtime #1654
  - Added `positive_only` objectives to `non_core_objectives` #1661
  - Fixed stacking argument `n_jobs` for IterativeAlgorithm #1706
  - Updated CatBoost estimators to return self in `.fit()` rather than the underlying model for consistency #1701
  - Added ability to initialize pipeline parameters in `AutoMLSearch` constructor #1676
- **Changes**
  - Added labeling to `graph_confusion_matrix` #1632
  - Rerunning search for `AutoMLSearch` results in a message thrown rather than failing the search, and removed `has_searched` property #1647
  - Changed tuner class to allow and ignore single parameter values as input #1686
  - Capped LightGBM version limit to remove bug in docs #1711
  - Removed support for *np.random.RandomState* in EvalML #1727
- **Documentation Changes**
  - Update Model Understanding in the user guide to include `visualize_decision_tree` #1678
  - Updated docs to include information about `AutoMLSearch` callback parameters and methods #1577
  - Updated docs to prompt users to install graphiz on Mac #1656
  - Added `infer_feature_types` to the `start.ipynb` guide #1700
  - Added multicollinearity data check to API reference and docs #1707
- Testing Changes

**Warning:**

**Breaking Changes**

- Removed `has_searched` property from `AutoMLSearch` #1647
- Components and pipelines return `Woodwork` data structures instead of `pandas` data structures #1668

- Removed support for *np.random.RandomState* in EvalML. Rather than passing `np.random.RandomState` as component and pipeline random_state values, we use int random_seed #1727

**v0.17.0 Dec. 29, 2020**

- **Enhancements**
  - Added `save_plot` that allows for saving figures from different backends #1588
  - Added `LightGBM Regressor` to regression components #1459
  - Added `visualize_decision_tree` for tree visualization with `decision_tree_data_from_estimator` and `decision_tree_data_from_pipeline` to reformat tree structure output #1511
  - Added *DFS Transformer* component into transformer components #1454
  - Added `MAPE` to the standard metrics for time series problems and update objectives #1510
  - Added `graph_prediction_vs_actual_over_time` and `get_prediction_vs_actual_over_time_data` to the model understanding module for time series problems #1483
  - Added a `ComponentGraph` class that will support future pipelines as directed acyclic graphs #1415
  - Updated data checks to accept `Woodwork` data structures #1481
  - Added parameter to `InvalidTargetDataCheck` to show only top unique values rather than all unique values #1485
  - Added multicollinearity data check #1515
  - Added baseline pipeline and components for time series regression problems #1496
  - Added more information to users about ensembling behavior in `AutoMLSearch` #1527
  - Add woodwork support for more utility and graph methods #1544
  - Changed `DateTimeFeaturizer` to encode features as int #1479
  - Return trained pipelines from `AutoMLSearch.best_pipeline` #1547
  - Added utility method so that users can set feature types without having to learn about Woodwork directly #1555
  - Added Linear Discriminant Analysis transformer for dimensionality reduction #1331
  - Added multiclass support for `partial_dependence` and `graph_partial_dependence` #1554
  - Added `TimeSeriesBinaryClassificationPipeline` and `TimeSeriesMulticlassClassificationPipelin` classes #1528
  - Added `make_data_splitter` method for easier automl data split customization #1568
  - Integrated `ComponentGraph` class into Pipelines for full non-linear pipeline support #1543
  - Update `AutoMLSearch` constructor to take training data instead of `search` and `add_to_leaderboard` #1597
  - Update `split_data` helper args #1597
  - Add problem type utils `is_regression`, `is_classification`, `is_timeseries` #1597
  - Rename `AutoMLSearch` `data_split` arg to `data_splitter` #1569
- **Fixes**

  - Fix AutoML not passing CV folds to `DefaultDataChecks` for usage by `ClassImbalanceDataCheck` #1619
  - Fix Windows CI jobs: install `numba` via conda, required for `shap` #1490
  - Added custom-index support for *reset-index-get_prediction_vs_actual_over_time_data* #1494
  - Fix `generate_pipeline_code` to account for boolean and None differences between Python and JSON #1524 #1531
  - Set max value for plotly and xgboost versions while we debug CI failures with newer versions #1532
  - Undo version pinning for plotly #1533
  - Fix ReadTheDocs build by updating the version of `setuptools` #1561
  - Set `random_state` of data splitter in AutoMLSearch to take int to keep consistency in the resulting splits #1579
  - Pin sklearn version while we work on adding support #1594
  - Pin pandas at <1.2.0 while we work on adding support #1609
  - Pin graphviz at < 0.16 while we work on adding support #1609
- **Changes**
  - Reverting `save_graph` #1550 to resolve kaleido build issues #1585
  - Update circleci badge to apply to `main` #1489
  - Added script to generate github markdown for releases #1487
  - Updated selection using pandas `dtypes` to selecting using Woodwork logical types #1551
  - Updated dependencies to fix `ImportError: cannot import name 'MaskedArray' from 'sklearn.utils.fixes'` error and to address Woodwork and Featuretool dependencies #1540
  - Made `get_prediction_vs_actual_data()` a public method #1553
  - Updated `Woodwork` version requirement to v0.0.7 #1560
  - Move data splitters from `evalml.automl.data_splitters` to `evalml.preprocessing.data_splitters` #1597
  - Rename "# Testing" in automl log output to "# Validation" #1597
- **Documentation Changes**
  - Added partial dependence methods to API reference #1537
  - Updated documentation for confusion matrix methods #1611
- **Testing Changes**
  - Set `n_jobs=1` in most unit tests to reduce memory #1505

**Warning:**

**Breaking Changes**

- Updated minimal dependencies: `numpy>=1.19.1`, `pandas>=1.1.0`, `scikit-learn>=0.23.1`, `scikit-optimize>=0.8.1`
- Updated `AutoMLSearch.best_pipeline` to return a trained pipeline. Pass in `train_best_pipeline=False` to AutoMLSearch in order to return an untrained pipeline.

- Pipeline component instances can no longer be iterated through using `Pipeline.component_graph` #1543
- Update `AutoMLSearch` constructor to take training data instead of `search` and `add_to_leaderboard` #1597
- Update `split_data` helper args #1597
- Move data splitters from `evalml.automl.data_splitters` to `evalml.preprocessing.data_splitters` #1597
- Rename `AutoMLSearch` `data_split` arg to `data_splitter` #1569

**v0.16.1 Dec. 1, 2020**

- **Enhancements**
  - Pin woodwork version to v0.0.6 to avoid breaking changes #1484
  - Updated `Woodwork` to >=0.0.5 in `core-requirements.txt` #1473
  - Removed `copy_dataframe` parameter for `Woodwork`, updated `Woodwork` to >=0.0.6 in `core-requirements.txt` #1478
  - Updated `detect_problem_type` to use `pandas.api.is_numeric_dtype` #1476
- **Changes**
  - Changed `make clean` to delete coverage reports as a convenience for developers #1464
  - Set `n_jobs=-1` by default for stacked ensemble components #1472
- **Documentation Changes**
  - Updated pipeline and component documentation and demos to use `Woodwork` #1466
- **Testing Changes**
  - Update dependency update checker to use everything from core and optional dependencies #1480

**v0.16.0 Nov. 24, 2020**

- **Enhancements**
  - Updated pipelines and `make_pipeline` to accept `Woodwork` inputs #1393
  - Updated components to accept `Woodwork` inputs #1423
  - Added ability to freeze hyperparameters for `AutoMLSearch` #1284
  - Added `Target Encoder` into transformer components #1401
  - Added callback for error handling in `AutoMLSearch` #1403
  - Added the index id to the `explain_predictions_best_worst` output to help users identify which rows in their data are included #1365
  - The top_k features displayed in `explain_predictions_*` functions are now determined by the magnitude of shap values as opposed to the `top_k` largest and smallest shap values. #1374
  - Added a problem type for time series regression #1386
  - Added a `is_defined_for_problem_type` method to `ObjectiveBase` #1386
  - Added a `random_state` parameter to `make_pipeline_from_components` function #1411
  - Added `DelayedFeaturesTransformer` #1396

  - Added a `TimeSeriesRegressionPipeline` class #1418
  - Removed `core-requirements.txt` from the package distribution #1429
  - Updated data check messages to include a *"code"* and *"details"* fields #1451, #1462
  - Added a `TimeSeriesSplit` data splitter for time series problems #1441
  - Added a `problem_configuration` parameter to AutoMLSearch #1457
- **Fixes**
  - Fixed `IndexError` raised in `AutoMLSearch` when `ensembling = True` but only one pipeline to iterate over #1397
  - Fixed stacked ensemble input bug and LightGBM warning and bug in `AutoMLSearch` #1388
  - Updated enum classes to show possible enum values as attributes #1391
  - Updated calls to `Woodwork`'s `to_pandas()` to `to_series()` and `to_dataframe()` #1428
  - Fixed bug in OHE where column names were not guaranteed to be unique #1349
  - Fixed bug with percent improvement of `ExpVariance` objective on data with highly skewed target #1467
  - Fix SimpleImputer error which occurs when all features are bool type #1215
- **Changes**
  - Changed `OutliersDataCheck` to return the list of columns, rather than rows, that contain outliers #1377
  - Simplified and cleaned output for Code Generation #1371
  - Reverted changes from #1337 #1409
  - Updated data checks to return dictionary of warnings and errors instead of a list #1448
  - Updated `AutoMLSearch` to pass `Woodwork` data structures to every pipeline (instead of pandas DataFrames) #1450
  - Update `AutoMLSearch` to default to `max_batches=1` instead of `max_iterations=5` #1452
  - Updated _evaluate_pipelines to consolidate side effects #1410
- **Documentation Changes**
  - Added description of CLA to contributing guide, updated description of draft PRs #1402
  - Updated documentation to include all data checks, `DataChecks`, and usage of data checks in AutoML #1412
  - Updated docstrings from `np.array` to `np.ndarray` #1417
  - Added section on stacking ensembles in AutoMLSearch documentation #1425
- **Testing Changes**
  - Removed `category_encoders` from test-requirements.txt #1373
  - Tweak codecov.io settings again to avoid flakes #1413
  - Modified `make lint` to check notebook versions in the docs #1431
  - Modified `make lint-fix` to standardize notebook versions in the docs #1431
  - Use new version of pull request Github Action for dependency check (#1443)
  - Reduced number of workers for tests to 4 #1447

**Warning:**

**Breaking Changes**

- The `top_k` and `top_k_features` parameters in `explain_predictions_*` functions now return `k` features as opposed to `2 * k` features #1374
- Renamed `problem_type` to `problem_types` in `RegressionObjective`, `BinaryClassificationObjective`, and `MulticlassClassificationObjective` #1319
- Data checks now return a dictionary of warnings and errors instead of a list #1448

**v0.15.0 Oct. 29, 2020**

- **Enhancements**
  - Added stacked ensemble component classes (`StackedEnsembleClassifier`, `StackedEnsembleRegressor`) #1134
  - Added stacked ensemble components to `AutoMLSearch` #1253
  - Added `DecisionTreeClassifier` and `DecisionTreeRegressor` to AutoML #1255
  - Added `graph_prediction_vs_actual` in `model_understanding` for regression problems #1252
  - Added parameter to `OneHotEncoder` to enable filtering for features to encode for #1249
  - Added percent-better-than-baseline for all objectives to automl.results #1244
  - Added `HighVarianceCVDataCheck` and replaced synonymous warning in `AutoMLSearch` #1254
  - Added *PCA Transformer* component for dimensionality reduction #1270
  - Added `generate_pipeline_code` and `generate_component_code` to allow for code generation given a pipeline or component instance #1306
  - Added `PCA Transformer` component for dimensionality reduction #1270
  - Updated `AutoMLSearch` to support `Woodwork` data structures #1299
  - Added cv_folds to `ClassImbalanceDataCheck` and added this check to `DefaultDataChecks` #1333
  - Make `max_batches` argument to `AutoMLSearch.search` public #1320
  - Added text support to automl search #1062
  - Added `_pipelines_per_batch` as a private argument to `AutoMLSearch` #1355
- **Fixes**
  - Fixed ML performance issue with ordered datasets: always shuffle data in automl's default CV splits #1265
  - Fixed broken `evalml info` CLI command #1293
  - Fixed `boosting type='rf'` for LightGBM Classifier, as well as `num_leaves` error #1302
  - Fixed bug in `explain_predictions_best_worst` where a custom index in the target variable would cause a `ValueError` #1318
  - Added stacked ensemble estimators to to `evalml.pipelines.__init__` file #1326

  - Fixed bug in OHE where calls to transform were not deterministic if `top_n` was less than the number of categories in a column #1324
  - Fixed LightGBM warning messages during AutoMLSearch #1342
  - Fix warnings thrown during AutoMLSearch in `HighVarianceCVDataCheck` #1346
  - Fixed bug where TrainingValidationSplit would return invalid location indices for dataframes with a custom index #1348
  - Fixed bug where the AutoMLSearch `random_state` was not being passed to the created pipelines #1321

- **Changes**
  - Allow `add_to_rankings` to be called before AutoMLSearch is called #1250
  - Removed Graphviz from test-requirements to add to requirements.txt #1327
  - Removed `max_pipelines` parameter from `AutoMLSearch` #1264
  - Include editable installs in all install make targets #1335
  - Made pip dependencies *featuretools* and *nlp_primitives* core dependencies #1062
  - Removed *PartOfSpeechCount* from *TextFeaturizer* transform primitives #1062
  - Added warning for `partial_dependency` when the feature includes null values #1352

- **Documentation Changes**
  - Fixed and updated code blocks in Release Notes #1243
  - Added DecisionTree estimators to API Reference #1246
  - Changed class inheritance display to flow vertically #1248
  - Updated cost-benefit tutorial to use a holdout/test set #1159
  - Added `evalml info` command to documentation #1293
  - Miscellaneous doc updates #1269
  - Removed conda pre-release testing from the release process document #1282
  - Updates to contributing guide #1310
  - Added Alteryx footer to docs with Twitter and Github link #1312
  - Added documentation for evalml installation for Python 3.6 #1322
  - Added documentation changes to make the API Docs easier to understand #1323
  - Fixed documentation for `feature_importance` #1353
  - Added tutorial for running *AutoML* with text data #1357
  - Added documentation for woodwork integration with automl search #1361

- **Testing Changes**
  - Added tests for `jupyter_check` to handle IPython #1256
  - Cleaned up `make_pipeline` tests to test for all estimators #1257
  - Added a test to check conda build after merge to main #1247
  - Removed code that was lacking codecov for `__main__.py` and unnecessary #1293
  - Codecov: round coverage up instead of down #1334

- Add DockerHub credentials to CI testing environment #1356
- Add DockerHub credentials to conda testing environment #1363

**Warning:**

**Breaking Changes**

- Renamed `LabelLeakageDataCheck` to `TargetLeakageDataCheck` #1319
- `max_pipelines` parameter has been removed from `AutoMLSearch`. Please use `max_iterations` instead. #1264
- `AutoMLSearch.search()` will now log a warning if the input is not a `Woodwork` data structure (`pandas`, `numpy`) #1299
- Make `max_batches` argument to `AutoMLSearch.search` public #1320
- Removed unused argument *feature_types* from AutoMLSearch.search #1062

**v0.14.1 Sep. 29, 2020**

- **Enhancements**
  - Updated partial dependence methods to support calculating numeric columns in a dataset with non-numeric columns #1150
  - Added `get_feature_names` on `OneHotEncoder` #1193
  - Added `detect_problem_type` to `problem_type/utils.py` to automatically detect the problem type given targets #1194
  - Added LightGBM to `AutoMLSearch` #1199
  - Updated `scikit-learn` and `scikit-optimize` to use latest versions - 0.23.2 and 0.8.1 respectively #1141
  - Added `__str__` and `__repr__` for pipelines and components #1218
  - Included internal target check for both training and validation data in `AutoMLSearch` #1226
  - Added `ProblemTypes.all_problem_types` helper to get list of supported problem types #1219
  - Added `DecisionTreeClassifier` and `DecisionTreeRegressor` classes #1223
  - Added `ProblemTypes.all_problem_types` helper to get list of supported problem types #1219
  - `DataChecks` can now be parametrized by passing a list of `DataCheck` classes and a parameter dictionary #1167
  - Added first CV fold score as validation score in `AutoMLSearch.rankings` #1221
  - Updated `flake8` configuration to enable linting on `__init__.py` files #1234
  - Refined `make_pipeline_from_components` implementation #1204
- **Fixes**
  - Updated GitHub URL after migration to Alteryx GitHub org #1207
  - Changed Problem Type enum to be more similar to the string name #1208
  - Wrapped call to scikit-learn's partial dependence method in a `try/finally` block #1232
- **Changes**
  - Added `allow_writing_files` as a named argument to CatBoost estimators. #1202

  - Added `solver` and `multi_class` as named arguments to `LogisticRegressionClassifier` #1202
  - Replaced pipeline's `._transform` method to evaluate all the preprocessing steps of a pipeline with `.compute_estimator_features` #1231
  - Changed default large dataset train/test splitting behavior #1205
- **Documentation Changes**
  - Included description of how to access the component instances and features for pipeline user guide #1163
  - Updated API docs to refer to target as "target" instead of "labels" for non-classification tasks and minor docs cleanup #1160
  - Added Class Imbalance Data Check to `api_reference.rst` #1190 #1200
  - Added pipeline properties to API reference #1209
  - Clarified what the objective parameter in AutoML is used for in AutoML API reference and AutoML user guide #1222
  - Updated API docs to include `skopt.space.Categorical` option for component hyperparameter range definition #1228
  - Added install documentation for `libomp` in order to use LightGBM on Mac #1233
  - Improved description of `max_iterations` in documentation #1212
  - Removed unused code from sphinx conf #1235
- Testing Changes

**Warning:**

**Breaking Changes**

- `DefaultDataChecks` now accepts a `problem_type` parameter that must be specified #1167
- Pipeline's `._transform` method to evaluate all the preprocessing steps of a pipeline has been replaced with `.compute_estimator_features` #1231
- `get_objectives` has been renamed to `get_core_objectives`. This function will now return a list of valid objective instances #1230

**v0.13.2 Sep. 17, 2020**

- **Enhancements**
  - Added `output_format` field to explain predictions functions #1107
  - Modified `get_objective` and `get_objectives` to be able to return any objective in `evalml.objectives` #1132
  - Added a `return_instance` boolean parameter to `get_objective` #1132
  - Added `ClassImbalanceDataCheck` to determine whether target imbalance falls below a given threshold #1135
  - Added label encoder to LightGBM for binary classification #1152
  - Added labels for the row index of confusion matrix #1154
  - Added `AutoMLSearch` object as another parameter in search callbacks #1156

  - Added the corresponding probability threshold for each point displayed in `graph_roc_curve` #1161
  - Added `__eq__` for `ComponentBase` and `PipelineBase` #1178
  - Added support for multiclass classification for `roc_curve` #1164
  - Added `categories` accessor to `OneHotEncoder` for listing the categories associated with a feature #1182
  - Added utility function to create pipeline instances from a list of component instances #1176
- **Fixes**
  - Fixed XGBoost column names for partial dependence methods #1104
  - Removed dead code validating column type from `TextFeaturizer` #1122
  - Fixed issue where `Imputer` cannot fit when there is None in a categorical or boolean column #1144
  - `OneHotEncoder` preserves the custom index in the input data #1146
  - Fixed representation for `ModelFamily` #1165
  - Removed duplicate `nbsphinx` dependency in `dev-requirements.txt` #1168
  - Users can now pass in any valid kwargs to all estimators #1157
  - Remove broken accessor `OneHotEncoder.get_feature_names` and unneeded base class #1179
  - Removed LightGBM Estimator from AutoML models #1186
- **Changes**
  - Pinned `scikit-optimize` version to 0.7.4 #1136
  - Removed `tqdm` as a dependency #1177
  - Added lightgbm version 3.0.0 to `latest_dependency_versions.txt` #1185
  - Rename `max_pipelines` to `max_iterations` #1169
- **Documentation Changes**
  - Fixed API docs for `AutoMLSearch` `add_result_callback` #1113
  - Added a step to our release process for pushing our latest version to conda-forge #1118
  - Added warning for missing ipywidgets dependency for using `PipelineSearchPlots` on Jupyterlab #1145
  - Updated `README.md` example to load demo dataset #1151
  - Swapped mapping of breast cancer targets in `model_understanding.ipynb` #1170
- **Testing Changes**
  - Added test confirming `TextFeaturizer` never outputs null values #1122
  - Changed Python version of `Update Dependencies` action to 3.8.x #1137
  - Fixed release notes check-in test for `Update Dependencies` actions #1172

**Warning:**

**Breaking Changes**

- `get_objective` will now return a class definition rather than an instance by default #1132
- Deleted `OPTIONS` dictionary in `evalml.objectives.utils.py` #1132
- If specifying an objective by string, the string must now match the objective's name field, case-insensitive #1132
- **Passing "Cost Benefit Matrix", "Fraud Cost", "Lead Scoring", "Mean Squared Log Error",** "Recall", "Recall Macro", "Recall Micro", "Recall Weighted", or "Root Mean Squared Log Error" to `AutoMLSearch` will now result in a `ValueError` rather than an `ObjectiveNotFoundError` #1132
- Search callbacks `start_iteration_callback` and `add_results_callback` have changed to include a copy of the AutoMLSearch object as a third parameter #1156
- Deleted `OneHotEncoder.get_feature_names` method which had been broken for a while, in favor of pipelines' `input_feature_names` #1179
- Deleted empty base class `CategoricalEncoder` which `OneHotEncoder` component was inheriting from #1176
- Results from `roc_curve` will now return as a list of dictionaries with each dictionary representing a class #1164
- `max_pipelines` now raises a `DeprecationWarning` and will be removed in the next release. `max_iterations` should be used instead. #1169

**v0.13.1 Aug. 25, 2020**

- **Enhancements**
  - Added Cost-Benefit Matrix objective for binary classification #1038
  - Split `fill_value` into `categorical_fill_value` and `numeric_fill_value` for Imputer #1019
  - Added `explain_predictions` and `explain_predictions_best_worst` for explaining multiple predictions with SHAP #1016
  - Added new LSA component for text featurization #1022
  - Added guide on installing with conda #1041
  - Added a "cost-benefit curve" util method to graph cost-benefit matrix scores vs. binary classification thresholds #1081
  - Standardized error when calling transform/predict before fit for pipelines #1048
  - Added `percent_better_than_baseline` to AutoML search rankings and full rankings table #1050
  - Added one-way partial dependence and partial dependence plots #1079
  - Added "Feature Value" column to prediction explanation reports. #1064
  - Added LightGBM classification estimator #1082, #1114
  - Added `max_batches` parameter to `AutoMLSearch` #1087
- **Fixes**
  - Updated `TextFeaturizer` component to no longer require an internet connection to run #1022
  - Fixed non-deterministic element of `TextFeaturizer` transformations #1022
  - Added a StandardScaler to all ElasticNet pipelines #1065

  - Updated cost-benefit matrix to normalize score #1099
  - Fixed logic in `calculate_percent_difference` so that it can handle negative values #1100
- **Changes**
  - Added `needs_fitting` property to `ComponentBase` #1044
  - Updated references to data types to use datatype lists defined in `evalml.utils.gen_utils` #1039
  - Remove maximum version limit for SciPy dependency #1051
  - Moved `all_components` and other component importers into runtime methods #1045
  - Consolidated graphing utility methods under `evalml.utils.graph_utils` #1060
  - Made slight tweaks to how `TextFeaturizer` uses `featuretools`, and did some refactoring of that and of LSA #1090
  - Changed `show_all_features` parameter into `importance_threshold`, which allows for thresholding feature importance #1097, #1103
- **Documentation Changes**
  - Update `setup.py` URL to point to the github repo #1037
  - Added tutorial for using the cost-benefit matrix objective #1088
  - Updated `model_understanding.ipynb` to include documentation for using plotly on Jupyter Lab #1108
- **Testing Changes**
  - Refactor CircleCI tests to use matrix jobs (#1043)
  - Added a test to check that all test directories are included in evalml package #1054

**Warning:**

**Breaking Changes**

- `confusion_matrix` and `normalize_confusion_matrix` have been moved to `evalml.utils` #1038
- All graph utility methods previously under `evalml.pipelines.graph_utils` have been moved to `evalml.utils.graph_utils` #1060

**v0.12.2 Aug. 6, 2020**

- **Enhancements**
  - Add save/load method to components #1023
  - Expose pickle `protocol` as optional arg to save/load #1023
  - Updated estimators used in AutoML to include ExtraTrees and ElasticNet estimators #1030
- Fixes
- **Changes**
  - Removed `DeprecationWarning` for `SimpleImputer` #1018
- **Documentation Changes**
  - Add note about version numbers to release process docs #1034

- **Testing Changes**
  - Test files are now included in the evalml package #1029

**v0.12.0 Aug. 3, 2020**

- **Enhancements**
  - Added string and categorical targets support for binary and multiclass pipelines and check for numeric targets for `DetectLabelLeakage` data check #932
  - Added clear exception for regression pipelines if target datatype is string or categorical #960
  - Added target column names and class labels in `predict` and `predict_proba` output for pipelines #951
  - Added `_compute_shap_values` and `normalize_values` to `pipelines/explanations` module #958
  - Added `explain_prediction` feature which explains single predictions with SHAP #974
  - Added Imputer to allow different imputation strategies for numerical and categorical dtypes #991
  - Added support for configuring logfile path using env var, and don't create logger if there are filesystem errors #975
  - Updated catboost estimators' default parameters and automl hyperparameter ranges to speed up fit time #998
- **Fixes**
  - Fixed ReadtheDocs warning failure regarding embedded gif #943
  - Removed incorrect parameter passed to pipeline classes in `_add_baseline_pipelines` #941
  - Added universal error for calling `predict`, `predict_proba`, `transform`, and `feature_importances` before fitting #969, #994
  - Made `TextFeaturizer` component and pip dependencies `featuretools` and `nlp_primitives` optional #976
  - Updated imputation strategy in automl to no longer limit impute strategy to `most_frequent` for all features if there are any categorical columns #991
  - Fixed `UnboundLocalError` for `cv_pipeline` when automl search errors #996
  - Fixed `Imputer` to reset dataframe index to preserve behavior expected from `SimpleImputer` #1009
- **Changes**
  - Moved `get_estimators` to `evalml.pipelines.components.utils` #934
  - Modified Pipelines to raise `PipelineScoreError` when they encounter an error during scoring #936
  - Moved `evalml.model_families.list_model_families` to `evalml.pipelines.components.allowed_model_families` #959
  - Renamed `DateTimeFeaturization` to `DateTimeFeaturizer` #977
  - Added check to stop search and raise an error if all pipelines in a batch return NaN scores #1015
- **Documentation Changes**
  - Updated `README.md` #963
  - Reworded message when errors are returned from data checks in search #982

  - Added section on understanding model predictions with `explain_prediction` to User Guide #981
  - Added a section to the user guide and api reference about how XGBoost and CatBoost are not fully supported. #992
  - Added custom components section in user guide #993
  - Updated FAQ section formatting #997
  - Updated release process documentation #1003
- **Testing Changes**
  - Moved `predict_proba` and `predict` tests regarding string / categorical targets to `test_pipelines.py` #972
  - Fixed dependency update bot by updating python version to 3.7 to avoid frequent github version updates #1002

**Warning:**

**Breaking Changes**

- `get_estimators` has been moved to `evalml.pipelines.components.utils` (previously was under `evalml.pipelines.utils`) #934
- Removed the `raise_errors` flag in AutoML search. All errors during pipeline evaluation will be caught and logged. #936
- `evalml.model_families.list_model_families` has been moved to `evalml.pipelines.components.allowed_model_families` #959
- `TextFeaturizer`: the `featuretools` and `nlp_primitives` packages must be installed after installing evalml in order to use this component #976
- Renamed `DateTimeFeaturization` to `DateTimeFeaturizer` #977

**v0.11.2 July 16, 2020**

- **Enhancements**
  - Added `NoVarianceDataCheck` to `DefaultDataChecks` #893
  - Added text processing and featurization component `TextFeaturizer` #913, #924
  - Added additional checks to `InvalidTargetDataCheck` to handle invalid target data types #929
  - `AutoMLSearch` will now handle `KeyboardInterrupt` and prompt user for confirmation #915
- **Fixes**
  - Makes automl results a read-only property #919
- **Changes**
  - Deleted static pipelines and refactored tests involving static pipelines, removed `all_pipelines()` and `get_pipelines()` #904
  - Moved `list_model_families` to `evalml.model_family.utils` #903
  - Updated `all_pipelines`, `all_estimators`, `all_components` to use the same mechanism for dynamically generating their elements #898
  - Rename `master` branch to `main` #918

  - Add pypi release github action #923
  - Updated `AutoMLSearch.search` stdout output and logging and removed tqdm progress bar #921
  - Moved automl config checks previously in `search()` to init #933
- **Documentation Changes**
  - Reorganized and rewrote documentation #937
  - Updated to use pydata sphinx theme #937
  - Updated docs to use `release_notes` instead of `changelog` #942
- **Testing Changes**
  - Cleaned up fixture names and usages in tests #895

**Warning:**

**Breaking Changes**

- `list_model_families` has been moved to `evalml.model_family.utils` (previously was under `evalml.pipelines.utils`) #903
- `get_estimators` has been moved to `evalml.pipelines.components.utils` (previously was under `evalml.pipelines.utils`) #934
- Static pipeline definitions have been removed, but similar pipelines can still be constructed via creating an instance of `PipelineBase` #904
- `all_pipelines()` and `get_pipelines()` utility methods have been removed #904

**v0.11.0 June 30, 2020**

- **Enhancements**
  - Added multiclass support for ROC curve graphing #832
  - Added preprocessing component to drop features whose percentage of NaN values exceeds a specified threshold #834
  - Added data check to check for problematic target labels #814
  - Added PerColumnImputer that allows imputation strategies per column #824
  - Added transformer to drop specific columns #827
  - Added support for `categories`, `handle_error`, and `drop` parameters in `OneHotEncoder` #830 #897
  - Added preprocessing component to handle DateTime columns featurization #838
  - Added ability to clone pipelines and components #842
  - Define getter method for component `parameters` #847
  - Added utility methods to calculate and graph permutation importances #860, #880
  - Added new utility functions necessary for generating dynamic preprocessing pipelines #852
  - Added kwargs to all components #863
  - Updated `AutoSearchBase` to use dynamically generated preprocessing pipelines #870
  - Added SelectColumns transformer #873

  - Added ability to evaluate additional pipelines for automl search #874
  - Added `default_parameters` class property to components and pipelines #879
  - Added better support for disabling data checks in automl search #892
  - Added ability to save and load AutoML objects to file #888
  - Updated `AutoSearchBase.get_pipelines` to return an untrained pipeline instance #876
  - Saved learned binary classification thresholds in automl results cv data dict #876
- **Fixes**
  - Fixed bug where SimpleImputer cannot handle dropped columns #846
  - Fixed bug where PerColumnImputer cannot handle dropped columns #855
  - Enforce requirement that builtin components save all inputted values in their parameters dict #847
  - Don't list base classes in `all_components` output #847
  - Standardize all components to output pandas data structures, and accept either pandas or numpy #853
  - Fixed rankings and full_rankings error when search has not been run #894
- **Changes**
  - Update `all_pipelines` and `all_components` to try initializing pipelines/components, and on failure exclude them #849
  - Refactor `handle_components` to `handle_components_class`, standardize to `ComponentBase` subclass instead of instance #850
  - Refactor "blacklist"/"whitelist" to "allow"/"exclude" lists #854
  - Replaced `AutoClassificationSearch` and `AutoRegressionSearch` with `AutoMLSearch` #871
  - Renamed feature_importances and permutation_importances methods to use singular names (feature_importance and permutation_importance) #883
  - Updated `automl` default data splitter to train/validation split for large datasets #877
  - Added open source license, update some repo metadata #887
  - Removed dead code in `_get_preprocessing_components` #896
- **Documentation Changes**
  - Fix some typos and update the EvalML logo #872
- **Testing Changes**
  - Update the changelog check job to expect the new branching pattern for the deps update bot #836
  - Check that all components output pandas datastructures, and can accept either pandas or numpy #853
  - Replaced `AutoClassificationSearch` and `AutoRegressionSearch` with `AutoMLSearch` #871

**Warning:**

**Breaking Changes**

- Pipelines' static `component_graph` field must contain either `ComponentBase` subclasses or `str`, instead of `ComponentBase` subclass instances #850
- Rename `handle_component` to `handle_component_class`. Now standardizes to `ComponentBase` subclasses instead of `ComponentBase` subclass instances #850
- Renamed automl's `cv` argument to `data_split` #877
- Pipelines' and classifiers' `feature_importances` is renamed `feature_importance`, `graph_feature_importances` is renamed `graph_feature_importance` #883
- Passing `data_checks=None` to automl search will not perform any data checks as opposed to default checks. #892
- Pipelines to search for in AutoML are now determined automatically, rather than using the statically-defined pipeline classes. #870
- Updated `AutoSearchBase.get_pipelines` to return an untrained pipeline instance, instead of one which happened to be trained on the final cross-validation fold #876

**v0.10.0 May 29, 2020**

- **Enhancements**
  - Added baseline models for classification and regression, add functionality to calculate baseline models before searching in AutoML #746
  - Port over highly-null guardrail as a data check and define `DefaultDataChecks` and `DisableDataChecks` classes #745
  - Update `Tuner` classes to work directly with pipeline parameters dicts instead of flat parameter lists #779
  - Add Elastic Net as a pipeline option #812
  - Added new Pipeline option `ExtraTrees` #790
  - Added precicion-recall curve metrics and plot for binary classification problems in `evalml.pipeline.graph_utils` #794
  - Update the default automl algorithm to search in batches, starting with default parameters for each pipeline and iterating from there #793
  - Added `AutoMLAlgorithm` class and `IterativeAlgorithm` impl, separated from `AutoSearchBase` #793
- **Fixes**
  - Update pipeline `score` to return `nan` score for any objective which throws an exception during scoring #787
  - Fixed bug introduced in #787 where binary classification metrics requiring predicted probabilities error in scoring #798
  - CatBoost and XGBoost classifiers and regressors can no longer have a learning rate of 0 #795
- **Changes**
  - Cleanup pipeline `score` code, and cleanup codecov #711
  - Remove `pass` for abstract methods for codecov #730
  - Added __str__ for AutoSearch object #675
  - Add util methods to graph ROC and confusion matrix #720

  - Refactor `AutoBase` to `AutoSearchBase` #758
  - Updated AutoBase with `data_checks` parameter, removed previous `detect_label_leakage` parameter, and added functionality to run data checks before search in AutoML #765
  - Updated our logger to use Python's logging utils #763
  - Refactor most of `AutoSearchBase._do_iteration` impl into `AutoSearchBase._evaluate` #762
  - Port over all guardrails to use the new DataCheck API #789
  - Expanded `import_or_raise` to catch all exceptions #759
  - Adds RMSE, MSLE, RMSLE as standard metrics #788
  - Don't allow `Recall` to be used as an objective for AutoML #784
  - Removed feature selection from pipelines #819
  - Update default estimator parameters to make automl search faster and more accurate #793
- **Documentation Changes**
  - Add instructions to freeze `master` on `release.md` #726
  - Update release instructions with more details #727 #733
  - Add objective base classes to API reference #736
  - Fix components API to match other modules #747
- **Testing Changes**
  - Delete codecov yml, use codecov.io's default #732
  - Added unit tests for fraud cost, lead scoring, and standard metric objectives #741
  - Update codecov client #782
  - Updated AutoBase __str__ test to include no parameters case #783
  - Added unit tests for `ExtraTrees` pipeline #790
  - If codecov fails to upload, fail build #810
  - Updated Python version of dependency action #816
  - Update the dependency update bot to use a suffix when creating branches #817

**Warning:**

**Breaking Changes**

- The `detect_label_leakage` parameter for AutoML classes has been removed and replaced by a `data_checks` parameter #765
- Moved ROC and confusion matrix methods from `evalml.pipeline.plot_utils` to `evalml.pipeline.graph_utils` #720
- `Tuner` classes require a pipeline hyperparameter range dict as an init arg instead of a space definition #779
- `Tuner.propose` and `Tuner.add` work directly with pipeline parameters dicts instead of flat parameter lists #779

- `PipelineBase.hyperparameters` and `custom_hyperparameters` use pipeline parameters dict format instead of being represented as a flat list #779
- All guardrail functions previously under `evalml.guardrails.utils` will be removed and replaced by data checks #789
- `Recall` disallowed as an objective for AutoML #784
- `AutoSearchBase` parameter `tuner` has been renamed to `tuner_class` #793
- `AutoSearchBase` parameter `possible_pipelines` and `possible_model_families` have been renamed to `allowed_pipelines` and `allowed_model_families` #793

**v0.9.0 Apr. 27, 2020**

- **Enhancements**
  - Added `Accuracy` as an standard objective #624
  - Added verbose parameter to load_fraud #560
  - Added Balanced Accuracy metric for binary, multiclass #612 #661
  - Added XGBoost regressor and XGBoost regression pipeline #666
  - Added `Accuracy` metric for multiclass #672
  - Added objective name in `AutoBase.describe_pipeline` #686
  - Added `DataCheck` and `DataChecks`, `Message` classes and relevant subclasses #739
- **Fixes**
  - Removed direct access to `cls.component_graph` #595
  - Add testing files to .gitignore #625
  - Remove circular dependencies from `Makefile` #637
  - Add error case for `normalize_confusion_matrix()` #640
  - Fixed `XGBoostClassifier` and `XGBoostRegressor` bug with feature names that contain [, ], or < #659
  - Update `make_pipeline_graph` to not accidentally create empty file when testing if path is valid #649
  - Fix pip installation warning about docsutils version, from boto dependency #664
  - Removed zero division warning for F1/precision/recall metrics #671
  - Fixed `summary` for pipelines without estimators #707
- **Changes**
  - Updated default objective for binary/multiclass classification to log loss #613
  - Created classification and regression pipeline subclasses and removed objective as an attribute of pipeline classes #405
  - Changed the output of `score` to return one dictionary #429
  - Created binary and multiclass objective subclasses #504
  - Updated objectives API #445
  - Removed call to `get_plot_data` from AutoML #615

  - Set `raise_error` to default to True for AutoML classes #638
  - Remove unnecessary "u" prefixes on some unicode strings #641
  - Changed one-hot encoder to return uint8 dtypes instead of ints #653
  - Pipeline `_name` field changed to `custom_name` #650
  - Removed `graphs.py` and moved methods into `PipelineBase` #657, #665
  - Remove s3fs as a dev dependency #664
  - Changed requirements-parser to be a core dependency #673
  - Replace `supported_problem_types` field on pipelines with `problem_type` attribute on base classes #678
  - Changed AutoML to only show best results for a given pipeline template in `rankings`, added `full_rankings` property to show all #682
  - Update `ModelFamily` values: don't list xgboost/catboost as classifiers now that we have regression pipelines for them #677
  - Changed AutoML's `describe_pipeline` to get problem type from pipeline instead #685
  - Standardize `import_or_raise` error messages #683
  - Updated argument order of objectives to align with sklearn's #698
  - Renamed `pipeline.feature_importance_graph` to `pipeline.graph_feature_importances` #700
  - Moved ROC and confusion matrix methods to `evalml.pipelines.plot_utils` #704
  - Renamed `MultiClassificationObjective` to `MulticlassClassificationObjective`, to align with pipeline naming scheme #715

- **Documentation Changes**
  - Fixed some sphinx warnings #593
  - Fixed docstring for `AutoClassificationSearch` with correct command #599
  - Limit readthedocs formats to pdf, not htmlzip and epub #594 #600
  - Clean up objectives API documentation #605
  - Fixed function on Exploring search results page #604
  - Update release process doc #567
  - `AutoClassificationSearch` and `AutoRegressionSearch` show inherited methods in API reference #651
  - Fixed improperly formatted code in breaking changes for changelog #655
  - Added configuration to treat Sphinx warnings as errors #660
  - Removed separate plotting section for pipelines in API reference #657, #665
  - Have leads example notebook load S3 files using https, so we can delete s3fs dev dependency #664
  - Categorized components in API reference and added descriptions for each category #663
  - Fixed Sphinx warnings about `BalancedAccuracy` objective #669
  - Updated API reference to include missing components and clean up pipeline docstrings #689
  - Reorganize API ref, and clarify pipeline sub-titles #688

  - Add and update preprocessing utils in API reference #687
  - Added inheritance diagrams to API reference #695
  - Documented which default objective AutoML optimizes for #699
  - Create seperate install page #701
  - Include more utils in API ref, like `import_or_raise` #704
  - Add more color to pipeline documentation #705
- **Testing Changes**
  - Matched install commands of `check_latest_dependencies` test and it's GitHub action #578
  - Added Github app to auto assign PR author as assignee #477
  - Removed unneeded conda installation of xgboost in windows checkin tests #618
  - Update graph tests to always use tmpfile dir #649
  - Changelog checkin test workaround for release PRs: If 'future release' section is empty of PR refs, pass check #658
  - Add changelog checkin test exception for `dep-update` branch #723

**Warning: Breaking Changes**

- Pipelines will now no longer take an objective parameter during instantiation, and will no longer have an objective attribute.
- `fit()` and `predict()` now use an optional `objective` parameter, which is only used in binary classification pipelines to fit for a specific objective.
- `score()` will now use a required `objectives` parameter that is used to determine all the objectives to score on. This differs from the previous behavior, where the pipeline's objective was scored on regardless.
- `score()` will now return one dictionary of all objective scores.
- `ROC` and `ConfusionMatrix` plot methods via `Auto(*).plot` have been removed by #615 and are replaced by `roc_curve` and `confusion_matrix` in `evamlm.pipelines.plot_utils` in #704
- `normalize_confusion_matrix` has been moved to `evalml.pipelines.plot_utils` #704
- Pipelines `_name` field changed to `custom_name`
- Pipelines `supported_problem_types` field is removed because it is no longer necessary #678
- Updated argument order of objectives' `objective_function` to align with sklearn #698
- `pipeline.feature_importance_graph` has been renamed to `pipeline.graph_feature_importances` in #700
- Removed unsupported `MSLE` objective #704

**v0.8.0 Apr. 1, 2020**

- **Enhancements**
  - Add normalization option and information to confusion matrix #484
  - Add util function to drop rows with NaN values #487
  - Renamed `PipelineBase.name` as `PipelineBase.summary` and redefined `PipelineBase.name` as class property #491

  - Added access to parameters in Pipelines with `PipelineBase.parameters` (used to be return of `PipelineBase.describe`) #501
  - Added `fill_value` parameter for `SimpleImputer` #509
  - Added functionality to override component hyperparameters and made pipelines take hyperparemeters from components #516
  - Allow `numpy.random.RandomState` for random_state parameters #556
- **Fixes**
  - Removed unused dependency `matplotlib`, and move `category_encoders` to test reqs #572
- **Changes**
  - Undo version cap in XGBoost placed in #402 and allowed all released of XGBoost #407
  - Support pandas 1.0.0 #486
  - Made all references to the logger static #503
  - Refactored `model_type` parameter for components and pipelines to `model_family` #507
  - Refactored `problem_types` for pipelines and components into `supported_problem_types` #515
  - Moved `pipelines/utils.save_pipeline` and `pipelines/utils.load_pipeline` to `PipelineBase.save` and `PipelineBase.load` #526
  - Limit number of categories encoded by `OneHotEncoder` #517
- **Documentation Changes**
  - Updated API reference to remove `PipelinePlot` and added moved `PipelineBase` plotting methods #483
  - Add code style and github issue guides #463 #512
  - Updated API reference for to surface class variables for pipelines and components #537
  - Fixed README documentation link #535
  - Unhid PR references in changelog #656
- **Testing Changes**
  - Added automated dependency check PR #482, #505
  - Updated automated dependency check comment #497
  - Have build_docs job use python executor, so that env vars are set properly #547
  - Added simple test to make sure `OneHotEncoder`'s top_n works with large number of categories #552
  - Run windows unit tests on PRs #557

**Warning: Breaking Changes**

- `AutoClassificationSearch` and `AutoRegressionSearch`'s `model_types` parameter has been refactored into `allowed_model_families`
- `ModelTypes` enum has been changed to `ModelFamily`
- Components and Pipelines now have a `model_family` field instead of `model_type`

- `get_pipelines` utility function now accepts `model_families` as an argument instead of `model_types`
- `PipelineBase.name` no longer returns structure of pipeline and has been replaced by `PipelineBase.summary`
- `PipelineBase.problem_types` and `Estimator.problem_types` has been renamed to `supported_problem_types`
- `pipelines/utils.save_pipeline` and `pipelines/utils.load_pipeline` moved to `PipelineBase.save` and `PipelineBase.load`

**v0.7.0 Mar. 9, 2020**

- **Enhancements**
  - Added emacs buffers to .gitignore #350
  - Add CatBoost (gradient-boosted trees) classification and regression components and pipelines #247
  - Added Tuner abstract base class #351
  - Added `n_jobs` as parameter for `AutoClassificationSearch` and `AutoRegressionSearch` #403
  - Changed colors of confusion matrix to shades of blue and updated axis order to match scikit-learn's #426
  - Added `PipelineBase .graph` and `.feature_importance_graph` methods, moved from previous location #423
  - Added support for python 3.8 #462
- **Fixes**
  - Fixed ROC and confusion matrix plots not being calculated if user passed own additional_objectives #276
  - Fixed ReadtheDocs `FileNotFoundError` exception for fraud dataset #439
- **Changes**
  - Added `n_estimators` as a tunable parameter for XGBoost #307
  - Remove unused parameter `ObjectiveBase.fit_needs_proba` #320
  - Remove extraneous parameter `component_type` from all components #361
  - Remove unused `rankings.csv` file #397
  - Downloaded demo and test datasets so unit tests can run offline #408
  - Remove `_needs_fitting` attribute from Components #398
  - Changed plot.feature_importance to show only non-zero feature importances by default, added optional parameter to show all #413
  - Refactored `PipelineBase` to take in parameter dictionary and moved pipeline metadata to class attribute #421
  - Dropped support for Python 3.5 #438
  - Removed unused `apply.py` file #449
  - Clean up `requirements.txt` to remove unused deps #451

  - Support installation without all required dependencies #459
- **Documentation Changes**
  - Update release.md with instructions to release to internal license key #354
- **Testing Changes**
  - Added tests for utils (and moved current utils to gen_utils) #297
  - Moved XGBoost install into it's own separate step on Windows using Conda #313
  - Rewind pandas version to before 1.0.0, to diagnose test failures for that version #325
  - Added dependency update checkin test #324
  - Rewind XGBoost version to before 1.0.0 to diagnose test failures for that version #402
  - Update dependency check to use a whitelist #417
  - Update unit test jobs to not install dev deps #455

**Warning: Breaking Changes**

- Python 3.5 will not be actively supported.

**v0.6.0 Dec. 16, 2019**

- **Enhancements**
  - Added ability to create a plot of feature importances #133
  - Add early stopping to AutoML using patience and tolerance parameters #241
  - Added ROC and confusion matrix metrics and plot for classification problems and introduce PipelineSearchPlots class #242
  - Enhanced AutoML results with search order #260
  - Added utility function to show system and environment information #300
- **Fixes**
  - Lower botocore requirement #235
  - Fixed decision_function calculation for `FraudCost` objective #254
  - Fixed return value of `Recall` metrics #264
  - Components return `self` on fit #289
- **Changes**
  - Renamed automl classes to `AutoRegressionSearch` and `AutoClassificationSearch` #287
  - Updating demo datasets to retain column names #223
  - Moving pipeline visualization to `PipelinePlot` class #228
  - Standarizing inputs as `pd.Dataframe` / `pd.Series` #130
  - Enforcing that pipelines must have an estimator as last component #277
  - Added `ipywidgets` as a dependency in `requirements.txt` #278
  - Added Random and Grid Search Tuners #240
- **Documentation Changes**

  - Adding class properties to API reference #244
  - Fix and filter FutureWarnings from scikit-learn #249, #257
  - Adding Linear Regression to API reference and cleaning up some Sphinx warnings #227
- **Testing Changes**
  - Added support for testing on Windows with CircleCI #226
  - Added support for doctests #233

**Warning: Breaking Changes**

- The `fit()` method for `AutoClassifier` and `AutoRegressor` has been renamed to `search()`.
- `AutoClassifier` has been renamed to `AutoClassificationSearch`
- `AutoRegressor` has been renamed to `AutoRegressionSearch`
- `AutoClassificationSearch.results` and `AutoRegressionSearch.results` now is a dictionary with `pipeline_results` and `search_order` keys. `pipeline_results` can be used to access a dictionary that is identical to the old `.results` dictionary. Whereas, `search_order` returns a list of the search order in terms of `pipeline_id`.
- Pipelines now require an estimator as the last component in `component_list`. Slicing pipelines now throws an `NotImplementedError` to avoid returning pipelines without an estimator.

**v0.5.2 Nov. 18, 2019**

- **Enhancements**
  - Adding basic pipeline structure visualization #211
- **Documentation Changes**
  - Added notebooks to build process #212

**v0.5.1 Nov. 15, 2019**

- **Enhancements**
  - Added basic outlier detection guardrail #151
  - Added basic ID column guardrail #135
  - Added support for unlimited pipelines with a `max_time` limit #70
  - Updated .readthedocs.yaml to successfully build #188
- **Fixes**
  - Removed MSLE from default additional objectives #203
  - Fixed `random_state` passed in pipelines #204
  - Fixed slow down in RFRegressor #206
- **Changes**
  - Pulled information for describe_pipeline from pipeline's new describe method #190
  - Refactored pipelines #108
  - Removed guardrails from Auto(*) #202, #208
- **Documentation Changes**

  - Updated documentation to show `max_time` enhancements #189
  - Updated release instructions for RTD #193
  - Added notebooks to build process #212
  - Added contributing instructions #213
  - Added new content #222

**v0.5.0 Oct. 29, 2019**

- **Enhancements**
  - Added basic one hot encoding #73
  - Use enums for model_type #110
  - Support for splitting regression datasets #112
  - Auto-infer multiclass classification #99
  - Added support for other units in `max_time` #125
  - Detect highly null columns #121
  - Added additional regression objectives #100
  - Show an interactive iteration vs. score plot when using fit() #134
- **Fixes**
  - Reordered `describe_pipeline` #94
  - Added type check for `model_type` #109
  - Fixed `s` units when setting string `max_time` #132
  - Fix objectives not appearing in API documentation #150
- **Changes**
  - Reorganized tests #93
  - Moved logging to its own module #119
  - Show progress bar history #111
  - Using `cloudpickle` instead of pickle to allow unloading of custom objectives #113
  - Removed render.py #154
- **Documentation Changes**
  - Update release instructions #140
  - Include additional_objectives parameter #124
  - Added Changelog #136
- **Testing Changes**
  - Code coverage #90
  - Added CircleCI tests for other Python versions #104
  - Added doc notebooks as tests #139
  - Test metadata for CircleCI and 2 core parallelism #137

**v0.4.1 Sep. 16, 2019**

- **Enhancements**
  - Added AutoML for classification and regressor using Autobase and Skopt #7 #9
  - Implemented standard classification and regression metrics #7
  - Added logistic regression, random forest, and XGBoost pipelines #7
  - Implemented support for custom objectives #15
  - Feature importance for pipelines #18
  - Serialization for pipelines #19
  - Allow fitting on objectives for optimal threshold #27
  - Added detect label leakage #31
  - Implemented callbacks #42
  - Allow for multiclass classification #21
  - Added support for additional objectives #79
- **Fixes**
  - Fixed feature selection in pipelines #13
  - Made `random_seed` usage consistent #45
- **Documentation Changes**
  - Documentation Changes
  - Added docstrings #6
  - Created notebooks for docs #6
  - Initialized readthedocs EvalML #6
  - Added favicon #38
- **Testing Changes**
  - Added testing for loading data #39

**v0.2.0 Aug. 13, 2019**

- **Enhancements**
  - Created fraud detection objective #4

**v0.1.0 July. 31, 2019**

- *First Release*
- **Enhancements**
  - Added lead scoring objecitve #1
  - Added basic classifier #1
- **Documentation Changes**
  - Initialized Sphinx for docs #1

# PYTHON MODULE INDEX

## e


evalml, 223
evalml.automl, 223
evalml.automl.automl_algorithm, 223
evalml.automl.automl_algorithm.automl_algorithm, 223
evalml.automl.automl_algorithm.default_algorithm, 225
evalml.automl.automl_algorithm.iterative_algorithm, 228
evalml.automl.automl_search, 257
evalml.automl.callbacks, 266
evalml.automl.engine, 238
evalml.automl.engine.cf_engine, 238
evalml.automl.engine.dask_engine, 241
evalml.automl.engine.engine_base, 244
evalml.automl.engine.sequential_engine, 248
evalml.automl.pipeline_search_plots, 267
evalml.automl.progress, 268
evalml.automl.utils, 269
evalml.data_checks, 285
evalml.data_checks.class_imbalance_data_check, 286
evalml.data_checks.data_check, 289
evalml.data_checks.data_check_action, 289
evalml.data_checks.data_check_action_code, 290
evalml.data_checks.data_check_action_option, 291
evalml.data_checks.data_check_message, 294
evalml.data_checks.data_check_message_code, 296
evalml.data_checks.data_check_message_type, 298
evalml.data_checks.data_checks, 299
evalml.data_checks.datetime_format_data_check, 299
evalml.data_checks.default_data_checks, 307
evalml.data_checks.id_columns_data_check, 308
evalml.data_checks.invalid_target_data_check, 312
evalml.data_checks.multicollinearity_data_check, 316
evalml.data_checks.no_variance_data_check, 318
evalml.data_checks.null_data_check, 321
evalml.data_checks.outliers_data_check, 325
evalml.data_checks.sparsity_data_check, 328
evalml.data_checks.target_distribution_data_check, 330
evalml.data_checks.target_leakage_data_check, 332
evalml.data_checks.ts_parameters_data_check, 334
evalml.data_checks.ts_splitting_data_check, 336
evalml.data_checks.uniqueness_data_check, 338
evalml.data_checks.utils, 340
evalml.demos, 387
evalml.demos.breast_cancer, 387
evalml.demos.churn, 388
evalml.demos.diabetes, 388
evalml.demos.fraud, 389
evalml.demos.weather, 389
evalml.demos.wine, 390
evalml.exceptions, 392
evalml.exceptions.exceptions, 392
evalml.model_family, 398
evalml.model_family.model_family, 398
evalml.model_family.utils, 399
evalml.model_understanding, 402
evalml.model_understanding.decision_boundary, 409
evalml.model_understanding.feature_explanations, 410
evalml.model_understanding.force_plots, 411
evalml.model_understanding.metrics, 413
evalml.model_understanding.partial_dependence_functions, 416
evalml.model_understanding.permutation_importance, 418
evalml.model_understanding.prediction_explanations, 402
evalml.model_understanding.prediction_explanations.explain

402
evalml.model_understanding.visualizations, 420
evalml.objectives, 440
evalml.objectives.binary_classification_objective, 440
evalml.objectives.cost_benefit_matrix, 443
evalml.objectives.fraud_cost, 446
evalml.objectives.lead_scoring, 449
evalml.objectives.multiclass_classification_objective, 452
evalml.objectives.objective_base, 454
evalml.objectives.regression_objective, 457
evalml.objectives.sensitivity_low_alert, 459
evalml.objectives.standard_metrics, 462
evalml.objectives.time_series_regression_objective, 527
evalml.objectives.utils, 530
evalml.pipelines, 619
evalml.pipelines.binary_classification_pipeline, 1552
evalml.pipelines.binary_classification_pipeline_mixin, 1560
evalml.pipelines.classification_pipeline, 1561
evalml.pipelines.component_graph, 1568
evalml.pipelines.components, 619
evalml.pipelines.components.component_base, 1323
evalml.pipelines.components.component_base_meta, 1326
evalml.pipelines.components.ensemble, 619
evalml.pipelines.components.ensemble.stacked_ensemble_base, 619
evalml.pipelines.components.ensemble.stacked_ensemble_classifier, 622
evalml.pipelines.components.ensemble.stacked_ensemble_regressor, 627
evalml.pipelines.components.estimators, 642
evalml.pipelines.components.estimators.classifier, 642
evalml.pipelines.components.estimators.classifiers.baseline_classifier, 642
evalml.pipelines.components.estimators.classifiers.catboost_classifier, 646
evalml.pipelines.components.estimators.classifiers.decision_tree_classifier, 650
evalml.pipelines.components.estimators.classifiers.elastic_net_classifier, 654
evalml.pipelines.components.estimators.classifiers.et_classifier, 658
evalml.pipelines.components.estimators.classifiers.knn_classifier, 663
evalml.pipelines.components.estimators.classifiers.lightgbm_classifier, 667
evalml.pipelines.components.estimators.classifiers.logistic_regression_classifier, 671
evalml.pipelines.components.estimators.classifiers.rf_classifier, 675
evalml.pipelines.components.estimators.classifiers.svm_classifier, 678
evalml.pipelines.components.estimators.classifiers.vowpal_wabbit_classifier, 682
evalml.pipelines.components.estimators.classifiers.xgboost_classifier, 692
evalml.pipelines.components.estimators.estimator, 853
evalml.pipelines.components.estimators.regressors, 741
evalml.pipelines.components.estimators.regressors.arima_regressor, 741
evalml.pipelines.components.estimators.regressors.baseline_regressor, 746
evalml.pipelines.components.estimators.regressors.catboost_regressor, 749
evalml.pipelines.components.estimators.regressors.decision_tree_regressor, 753
evalml.pipelines.components.estimators.regressors.elastic_net_regressor, 758
evalml.pipelines.components.estimators.regressors.et_regressor, 761
evalml.pipelines.components.estimators.regressors.exponential_smoothing_regressor, 766
evalml.pipelines.components.estimators.regressors.lightgbm_regressor, 770
evalml.pipelines.components.estimators.regressors.linear_regressor, 774
evalml.pipelines.components.estimators.regressors.prophet_regressor, 777
evalml.pipelines.components.estimators.regressors.rf_regressor, 782
evalml.pipelines.components.estimators.regressors.svm_regressor, 786
evalml.pipelines.components.estimators.regressors.time_series_baseline_estimator, 790
evalml.pipelines.components.estimators.regressors.vowpal_wabbit_regressor, 793
evalml.pipelines.components.estimators.regressors.xgboost_regressor, 797
evalml.pipelines.components.transformers, 956
evalml.pipelines.components.transformers.column_selectors, 1201
evalml.pipelines.components.transformers.dimensionality_reduction.lda_transformer, 957
evalml.pipelines.components.transformers.dimensionality_reduction.pca_transformer, 957
evalml.pipelines.components.transformers.dimensionality_reduction.svd_transformer, 960

evalml.pipelines.components.transformers.encoders, 968
evalml.pipelines.components.transformers.encoders.label_encoder, 968
evalml.pipelines.components.transformers.encoders.onehot_encoder, 971
evalml.pipelines.components.transformers.encoders.ordinal_encoder, 976
evalml.pipelines.components.transformers.encoders.target_encoder, 980
evalml.pipelines.components.transformers.feature_selection, 996
evalml.pipelines.components.transformers.feature_selection.feature_selector, 996
evalml.pipelines.components.transformers.feature_selection.recursive_feature_elimination_selector, 999
evalml.pipelines.components.transformers.feature_selection.classwise_feature_selector, 1009
evalml.pipelines.components.transformers.feature_selection.regression_feature_selector, 1013
evalml.pipelines.components.transformers.imputers, 1032
evalml.pipelines.components.transformers.imputers.base_imputer, 1032
evalml.pipelines.components.transformers.imputers.knn_imputer, 1036
evalml.pipelines.components.transformers.imputers.per_column_imputer, 1039
evalml.pipelines.components.transformers.imputers.simple_imputer, 1042
evalml.pipelines.components.transformers.imputers.target_imputer, 1045
evalml.pipelines.components.transformers.imputers.time_series_imputer, 1048
evalml.pipelines.components.transformers.preprocessing, 1068
evalml.pipelines.components.transformers.preprocessing.datetime_featurizer, 1068
evalml.pipelines.components.transformers.preprocessing.delete_columns_transformer, 1071
evalml.pipelines.components.transformers.preprocessing.drop_nan_rows_transformer, 1076
evalml.pipelines.components.transformers.preprocessing.drop_rows_transformer, 1079
evalml.pipelines.components.transformers.preprocessing.drop_uniformly_random_rows_transformer, 1082
evalml.pipelines.components.transformers.preprocessing.featuretools, 1085
evalml.pipelines.components.transformers.preprocessing.log_transformer, 1088
evalml.pipelines.components.transformers.preprocessing.lsa, 1091
evalml.pipelines.components.transformers.preprocessing.natural_language_featurizer, 1094
evalml.pipelines.components.transformers.preprocessing.polynomial_decomposer, 1097
evalml.pipelines.components.transformers.preprocessing.replace_nan_transformer, 1103
evalml.pipelines.components.transformers.preprocessing.stl_decomposer, 1106
evalml.pipelines.components.transformers.preprocessing.text_transformer, 1112
evalml.pipelines.components.transformers.preprocessing.time_series_baseline_featurizer, 1115
evalml.pipelines.components.transformers.preprocessing.time_series_featurizer, 1119
evalml.pipelines.components.transformers.preprocessing.transformer_utils, 1122
evalml.pipelines.components.transformers.samplers, 1180
evalml.pipelines.components.transformers.samplers.base_sampler, 1180
evalml.pipelines.components.transformers.samplers.oversampler, 1183
evalml.pipelines.components.transformers.samplers.undersampler, 1186
evalml.pipelines.components.transformers.scalers, 1195
evalml.pipelines.components.transformers.scalers.standard_scaler, 1195
evalml.pipelines.components.transformer, 1211
evalml.pipelines.components.utils, 1327
evalml.pipelines.multiclass_classification_pipeline, 1574
evalml.pipelines.pipeline_base, 1581
evalml.pipelines.pipeline_meta, 1587
evalml.pipelines.regression_pipeline, 1589
evalml.pipelines.time_series_classification_pipelines, 1595
evalml.pipelines.time_series_pipeline_base, 1619
evalml.pipelines.time_series_regression_pipeline, 1626
evalml.pipelines.utils, 1635
evalml.preprocessing, 1852
evalml.preprocessing.data_splitters, 1852
evalml.preprocessing.data_splitters.no_split, 1853
evalml.preprocessing.data_splitters.sk_splitters, 1854
evalml.preprocessing.data_splitters.time_series_split, 1856
evalml.preprocessing.data_splitters.training_validation_split, 1858
evalml.preprocessing.utils, 1866
evalml.problem_types, 1875
evalml.problem_types.problem_types, 1876

evalml.problem_types.utils, 1877
evalml.tuners, 1883
evalml.tuners.grid_search_tuner, 1883
evalml.tuners.random_search_tuner, 1885
evalml.tuners.skopt_tuner, 1887
evalml.tuners.tuner, 1889
evalml.tuners.tuner_exceptions, 1891
evalml.utils, 1897
evalml.utils.base_meta, 1897
evalml.utils.cli_utils, 1898
evalml.utils.gen_utils, 1900
evalml.utils.logger, 1906
evalml.utils.nullable_type_utils, 1907
evalml.utils.update_checker, 1907
evalml.utils.woodwork_utils, 1907

# INDEX

## A


abs_error() (*in module evalml.model_understanding.prediction_explanations.explainers*), 403
AccuracyBinary (*class in evalml.objectives*), 534
AccuracyBinary (*class in evalml.objectives.standard_metrics*), 463
AccuracyMulticlass (*class in evalml.objectives*), 536
AccuracyMulticlass (*class in evalml.objectives.standard_metrics*), 465
add() (*evalml.tuners.grid_search_tuner.GridSearchTuner method*), 1884
add() (*evalml.tuners.GridSearchTuner method*), 1892
add() (*evalml.tuners.random_search_tuner.RandomSearchTuner method*), 1886
add() (*evalml.tuners.RandomSearchTuner method*), 1894
add() (*evalml.tuners.skopt_tuner.SKOptTuner method*), 1888
add() (*evalml.tuners.SKOptTuner method*), 1895
add() (*evalml.tuners.Tuner method*), 1896
add() (*evalml.tuners.tuner.Tuner method*), 1890
add_result() (*evalml.automl.automl_algorithm.automl_algorithm.AutoMLAlgorithm method*), 224
add_result() (*evalml.automl.automl_algorithm.AutoMLAlgorithm method*), 232
add_result() (*evalml.automl.automl_algorithm.default_algorithm.DefaultAlgorithm method*), 227
add_result() (*evalml.automl.automl_algorithm.DefaultAlgorithm method*), 235
add_result() (*evalml.automl.automl_algorithm.iterative_algorithm.IterativeAlgorithm method*), 230
add_result() (*evalml.automl.automl_algorithm.IterativeAlgorithm method*), 237
add_to_rankings() (*evalml.automl.automl_search.AutoMLSearch method*), 261
add_to_rankings() (*evalml.automl.AutoMLSearch method*), 277
add_to_rankings() (*evalml.AutoMLSearch method*), 1917
all_components() (*in module evalml.pipelines.components.utils*), 1328
all_parameter_types() (*evalml.data_checks.data_check_action_option.DCAOParameterT method*), 294
all_parameter_types() (*evalml.data_checks.DCAOParameterType method*), 359
all_problem_types() (*evalml.problem_types.problem_types.ProblemTypes method*), 1876
all_problem_types() (*evalml.problem_types.ProblemTypes method*), 1883
allowed_model_families() (*in module evalml.pipelines.components.utils*), 1328
are_datasets_separated_by_gap_time_index() (*in module evalml.utils.gen_utils*), 1902
are_ts_parameters_valid_for_split() (*in module evalml.utils.gen_utils*), 1902
ARIMARegressor (*class in evalml.pipelines*), 1642
ARIMARegressor (*class in evalml.pipelines.components*), 1337
ARIMARegressor (*class in evalml.pipelines.components.estimators*), 859
ARIMARegressor (*class in evalml.pipelines.components.estimators.regressors*), 801
ARIMARegressor (*class in evalml.pipelines.components.estimators.regressors.arima_regress*), 741
AUC (*class in evalml.objectives*), 538
AUC (*class in evalml.objectives.standard_metrics*), 467
AUCMacro (*class in evalml.objectives*), 540
AUCMacro (*class in evalml.objectives.standard_metrics*), 469
AUCMicro (*class in evalml.objectives*), 542
AUCMicro (*class in evalml.objectives.standard_metrics*), 471
AUCWeighted (*class in evalml.objectives*), 544
AUCWeighted (*class in evalml.objectives.standard_metrics*), 473
AutoMLAlgorithm (*class in*

*evalml.automl.automl_algorithm*), 232
AutoMLAlgorithm (*class in evalml.automl.automl_algorithm.automl_algorithm*), 223
AutoMLAlgorithmException, 225, 233
AutoMLConfig (*in module evalml.automl.utils*), 270
AutoMLSearch (*class in evalml*), 1914
AutoMLSearch (*class in evalml.automl*), 273
AutoMLSearch (*class in evalml.automl.automl_search*), 258
AutoMLSearchException, 392, 395

## B

BalancedAccuracyBinary (*class in evalml.objectives*), 545
BalancedAccuracyBinary (*class in evalml.objectives.standard_metrics*), 474
BalancedAccuracyMulticlass (*class in evalml.objectives*), 548
BalancedAccuracyMulticlass (*class in evalml.objectives.standard_metrics*), 477
BaselineClassifier (*class in evalml.pipelines.components*), 1340
BaselineClassifier (*class in evalml.pipelines.components.estimators*), 862
BaselineClassifier (*class in evalml.pipelines.components.estimators.classifiers*), 696
BaselineClassifier (*class in evalml.pipelines.components.estimators.classifiers.baseline_classifier*), 643
BaselineRegressor (*class in evalml.pipelines.components*), 1343
BaselineRegressor (*class in evalml.pipelines.components.estimators*), 865
BaselineRegressor (*class in evalml.pipelines.components.estimators.regressors*), 805
BaselineRegressor (*class in evalml.pipelines.components.estimators.regressors.baseline_regressor*), 746
BaseMeta (*class in evalml.utils.base_meta*), 1898
BaseSampler (*class in evalml.pipelines.components.transformers.samplers.base_sampler*), 1180
batch_number (*evalml.automl.automl_algorithm.automl_algorithm.AutoMLAlgorithm property*), 224
batch_number (*evalml.automl.automl_algorithm.AutoMLAlgorithm property*), 233
batch_number (*evalml.automl.automl_algorithm.default_algorithm.DefaultAlgorithm property*), 227
batch_number (*evalml.automl.automl_algorithm.DefaultAlgorithm property*), 235
batch_number (*evalml.automl.automl_algorithm.iterative_algorithm.IterativeAlgorithm property*), 231
batch_number (*evalml.automl.automl_algorithm.IterativeAlgorithm property*), 238
best_pipeline (*evalml.automl.automl_search.AutoMLSearch property*), 261
best_pipeline (*evalml.automl.AutoMLSearch property*), 277
best_pipeline (*evalml.AutoMLSearch property*), 1917
binary_objective_vs_threshold() (*in module evalml.model_understanding*), 428
binary_objective_vs_threshold() (*in module evalml.model_understanding.visualizations*), 421
BinaryClassificationObjective (*class in evalml.objectives*), 549
BinaryClassificationObjective (*class in evalml.objectives.binary_classification_objective*), 440
BinaryClassificationPipeline (*class in evalml.pipelines*), 1645
BinaryClassificationPipeline (*class in evalml.pipelines.binary_classification_pipeline*), 1553
BinaryClassificationPipelineMixin (*class in evalml.pipelines.binary_classification_pipeline_mixin*), 1561
build_engine_from_str() (*in module evalml.automl.automl_search*), 264
build_prophet_df() (*evalml.pipelines.components.estimators.ProphetRe static method*), 923
build_prophet_df() (*evalml.pipelines.components.estimators.regressors static method*), 780
build_prophet_df() (*evalml.pipelines.components.estimators.regressors static method*), 835
build_prophet_df() (*evalml.pipelines.components.ProphetRegressor static method*), 1457
build_prophet_df() (*evalml.pipelines.ProphetRegressor static method*), 1749

## C

calculate_percent_difference() (*evalml.objectives.AccuracyBinary class method*), 535
calculate_percent_difference() (*evalml.objectives.AccuracyMulticlass class method*), 537
calculate_percent_difference() (*evalml.objectives.AUC class method*), 539
calculate_percent_difference() (*evalml.objectives.AUCMacro class method*), 541

calculate_percent_difference() (*evalml.objectives.AUCMicro class method*), 543
calculate_percent_difference() (*evalml.objectives.AUCWeighted class method*), 544
calculate_percent_difference() (*evalml.objectives.BalancedAccuracyBinary class method*), 546
calculate_percent_difference() (*evalml.objectives.BalancedAccuracyMulticlass class method*), 548
calculate_percent_difference() (*evalml.objectives.binary_classification_objective.BinaryClassificationObjective class method*), 441
calculate_percent_difference() (*evalml.objectives.BinaryClassificationObjective class method*), 550
calculate_percent_difference() (*evalml.objectives.cost_benefit_matrix.CostBenefitMatrix class method*), 444
calculate_percent_difference() (*evalml.objectives.CostBenefitMatrix class method*), 553
calculate_percent_difference() (*evalml.objectives.ExpVariance class method*), 555
calculate_percent_difference() (*evalml.objectives.F1 class method*), 557
calculate_percent_difference() (*evalml.objectives.F1Macro class method*), 559
calculate_percent_difference() (*evalml.objectives.F1Micro class method*), 561
calculate_percent_difference() (*evalml.objectives.F1Weighted class method*), 562
calculate_percent_difference() (*evalml.objectives.fraud_cost.FraudCost class method*), 447
calculate_percent_difference() (*evalml.objectives.FraudCost class method*), 564
calculate_percent_difference() (*evalml.objectives.Gini class method*), 568
calculate_percent_difference() (*evalml.objectives.lead_scoring.LeadScoring class method*), 450
calculate_percent_difference() (*evalml.objectives.LeadScoring class method*), 570
calculate_percent_difference() (*evalml.objectives.LogLossBinary class method*), 573
calculate_percent_difference() (*evalml.objectives.LogLossMulticlass class method*), 575
calculate_percent_difference() (*evalml.objectives.MAE class method*), 577
calculate_percent_difference() (*evalml.objectives.MAPE class method*), 578
calculate_percent_difference() (*evalml.objectives.MaxError class method*), 580
calculate_percent_difference() (*evalml.objectives.MCCBinary class method*), 582
calculate_percent_difference() (*evalml.objectives.MCCMulticlass class method*), 584
calculate_percent_difference() (*evalml.objectives.MeanSquaredLogError class method*), 586
calculate_percent_difference() (*evalml.objectives.MedianAE class method*), 587
calculate_percent_difference() (*evalml.objectives.MSE class method*), 589
calculate_percent_difference() (*evalml.objectives.multiclass_classification_objective.MulticlassC class method*), 452
calculate_percent_difference() (*evalml.objectives.MulticlassClassificationObjective class method*), 590
calculate_percent_difference() (*evalml.objectives.objective_base.ObjectiveBase class method*), 455
calculate_percent_difference() (*evalml.objectives.ObjectiveBase class method*), 593
calculate_percent_difference() (*evalml.objectives.Precision class method*), 595
calculate_percent_difference() (*evalml.objectives.PrecisionMacro class method*), 597
calculate_percent_difference() (*evalml.objectives.PrecisionMicro class method*), 599
calculate_percent_difference() (*evalml.objectives.PrecisionWeighted class method*), 601
calculate_percent_difference() (*evalml.objectives.R2 class method*), 602
calculate_percent_difference() (*evalml.objectives.Recall class method*),

604
calculate_percent_difference() (*evalml.objectives.RecallMacro class method*), 606
calculate_percent_difference() (*evalml.objectives.RecallMicro class method*), 608
calculate_percent_difference() (*evalml.objectives.RecallWeighted class method*), 610
calculate_percent_difference() (*evalml.objectives.regression_objective.RegressionObjective class method*), 457
calculate_percent_difference() (*evalml.objectives.RegressionObjective class method*), 611
calculate_percent_difference() (*evalml.objectives.RootMeanSquaredError class method*), 614
calculate_percent_difference() (*evalml.objectives.RootMeanSquaredLogError class method*), 615
calculate_percent_difference() (*evalml.objectives.sensitivity_low_alert.SensitivityLowAlert class method*), 460
calculate_percent_difference() (*evalml.objectives.SensitivityLowAlert class method*), 617
calculate_percent_difference() (*evalml.objectives.standard_metrics.AccuracyBinary class method*), 464
calculate_percent_difference() (*evalml.objectives.standard_metrics.AccuracyMulticlass class method*), 466
calculate_percent_difference() (*evalml.objectives.standard_metrics.AUC class method*), 468
calculate_percent_difference() (*evalml.objectives.standard_metrics.AUCMacro class method*), 470
calculate_percent_difference() (*evalml.objectives.standard_metrics.AUCMicro class method*), 472
calculate_percent_difference() (*evalml.objectives.standard_metrics.AUCWeighted class method*), 473
calculate_percent_difference() (*evalml.objectives.standard_metrics.BalancedAccuracyBinary class method*), 475
calculate_percent_difference() (*evalml.objectives.standard_metrics.BalancedAccuracyMulticlass class method*), 477
calculate_percent_difference() (*evalml.objectives.standard_metrics.ExpVariance class method*), 479
calculate_percent_difference() (*evalml.objectives.standard_metrics.F1 class method*), 481
calculate_percent_difference() (*evalml.objectives.standard_metrics.F1Macro class method*), 483
calculate_percent_difference() (*evalml.objectives.standard_metrics.F1Micro class method*), 485
calculate_percent_difference() (*evalml.objectives.standard_metrics.F1Weighted class method*), 486
calculate_percent_difference() (*evalml.objectives.standard_metrics.Gini class method*), 488
calculate_percent_difference() (*evalml.objectives.standard_metrics.LogLossBinary class method*), 490
calculate_percent_difference() (*evalml.objectives.standard_metrics.LogLossMulticlass class method*), 492
calculate_percent_difference() (*evalml.objectives.standard_metrics.MAE class method*), 494
calculate_percent_difference() (*evalml.objectives.standard_metrics.MAPE class method*), 496
calculate_percent_difference() (*evalml.objectives.standard_metrics.MaxError class method*), 498
calculate_percent_difference() (*evalml.objectives.standard_metrics.MCCBinary class method*), 499
calculate_percent_difference() (*evalml.objectives.standard_metrics.MCCMulticlass class method*), 502
calculate_percent_difference() (*evalml.objectives.standard_metrics.MeanSquaredLogError class method*), 503
calculate_percent_difference() (*evalml.objectives.standard_metrics.MedianAE class method*), 505
calculate_percent_difference() (*evalml.objectives.standard_metrics.MSE class method*), 507
calculate_percent_difference() (*evalml.objectives.standard_metrics.Precision class method*), 509
calculate_percent_difference() (*evalml.objectives.standard_metrics.PrecisionMacro class method*), 511
calculate_percent_difference() (*evalml.objectives.standard_metrics.PrecisionMicro

class method), 512
calculate_percent_difference() (evalml.objectives.standard_metrics.PrecisionWeighted class method), 514
calculate_percent_difference() (evalml.objectives.standard_metrics.R2 class method), 516
calculate_percent_difference() (evalml.objectives.standard_metrics.Recall class method), 517
calculate_percent_difference() (evalml.objectives.standard_metrics.RecallMacro class method), 520
calculate_percent_difference() (evalml.objectives.standard_metrics.RecallMicro class method), 521
calculate_percent_difference() (evalml.objectives.standard_metrics.RecallWeighted class method), 523
calculate_percent_difference() (evalml.objectives.standard_metrics.RootMeanSquaredErrorLog class method), 525
calculate_percent_difference() (evalml.objectives.standard_metrics.RootMeanSquaredLogError class method), 526
calculate_percent_difference() (evalml.objectives.time_series_regression_objective.TimeSeriesRegressionObjective class method), 528
calculate_permutation_importance() (in module evalml.model_understanding), 428
calculate_permutation_importance() (in module evalml.model_understanding.permutation_importance), 419
calculate_permutation_importance_one_column() (in module evalml.model_understanding), 428
calculate_permutation_importance_one_column() (in module evalml.model_understanding.permutation_importance), 419
can_optimize_threshold (evalml.objectives.AccuracyBinary property), 535
can_optimize_threshold (evalml.objectives.AUC property), 539
can_optimize_threshold (evalml.objectives.BalancedAccuracyBinary property), 547
can_optimize_threshold (evalml.objectives.binary_classification_objective.BinaryClassificationObjective property), 441
can_optimize_threshold (evalml.objectives.BinaryClassificationObjective property), 550
can_optimize_threshold (evalml.objectives.cost_benefit_matrix.CostBenefitMatrix property), 445
can_optimize_threshold (evalml.objectives.CostBenefitMatrix property), 553
can_optimize_threshold (evalml.objectives.F1 property), 557
can_optimize_threshold (evalml.objectives.fraud_cost.FraudCost property), 447
can_optimize_threshold (evalml.objectives.FraudCost property), 564
can_optimize_threshold (evalml.objectives.Gini property), 569
can_optimize_threshold (evalml.objectives.lead_scoring.LeadScoring property), 450
can_optimize_threshold (evalml.objectives.LeadScoring property), 571
can_optimize_threshold (evalml.objectives.LogLossBinary property), 573
can_optimize_threshold (evalml.objectives.MCCBinary property), 582
can_optimize_threshold (evalml.objectives.Precision property), 596
can_optimize_threshold (evalml.objectives.Recall property), 605
can_optimize_threshold (evalml.objectives.sensitivity_low_alert.SensitivityLowAlert property), 460
can_optimize_threshold (evalml.objectives.SensitivityLowAlert property), 617
can_optimize_threshold (evalml.objectives.standard_metrics.AccuracyBinary property), 464
can_optimize_threshold (evalml.objectives.standard_metrics.AUC property), 468
can_optimize_threshold (evalml.objectives.standard_metrics.BalancedAccuracyBinary property), 476
can_optimize_threshold (evalml.objectives.standard_metrics.F1 property), 481
can_optimize_threshold (evalml.objectives.standard_metrics.Gini property), 488
can_optimize_threshold (evalml.objectives.standard_metrics.LogLossBinary property), 491

can_optimize_threshold (*evalml.objectives.standard_metrics.MCCBinary property*), 500
can_optimize_threshold (*evalml.objectives.standard_metrics.Precision property*), 509
can_optimize_threshold (*evalml.objectives.standard_metrics.Recall property*), 518
can_tune_threshold_with_objective() (*evalml.pipelines.binary_classification_pipeline.BinaryClassificationPipeline method*), 1556
can_tune_threshold_with_objective() (*evalml.pipelines.BinaryClassificationPipeline method*), 1648
can_tune_threshold_with_objective() (*evalml.pipelines.classification_pipeline.ClassificationPipeline method*), 1563
can_tune_threshold_with_objective() (*evalml.pipelines.ClassificationPipeline method*), 1660
can_tune_threshold_with_objective() (*evalml.pipelines.multiclass_classification_pipeline.MulticlassClassificationPipeline method*), 1576
can_tune_threshold_with_objective() (*evalml.pipelines.MulticlassClassificationPipeline method*), 1728
can_tune_threshold_with_objective() (*evalml.pipelines.pipeline_base.PipelineBase method*), 1583
can_tune_threshold_with_objective() (*evalml.pipelines.PipelineBase method*), 1743
can_tune_threshold_with_objective() (*evalml.pipelines.regression_pipeline.RegressionPipeline method*), 1591
can_tune_threshold_with_objective() (*evalml.pipelines.RegressionPipeline method*), 1759
can_tune_threshold_with_objective() (*evalml.pipelines.time_series_classification_pipeline.TimeSeriesBinaryClassificationPipeline method*), 1598
can_tune_threshold_with_objective() (*evalml.pipelines.time_series_classification_pipeline.TimeSeriesClassificationPipeline method*), 1606
can_tune_threshold_with_objective() (*evalml.pipelines.time_series_classification_pipeline.TimeSeriesMulticlassClassificationPipeline method*), 1613
can_tune_threshold_with_objective() (*evalml.pipelines.time_series_pipeline_base.TimeSeriesPipelineBase method*), 1621
can_tune_threshold_with_objective() (*evalml.pipelines.time_series_regression_pipeline.TimeSeriesRegressionPipeline method*), 1629
can_tune_threshold_with_objective() (*evalml.pipelines.TimeSeriesBinaryClassificationPipeline method*), 1796
can_tune_threshold_with_objective() (*evalml.pipelines.TimeSeriesClassificationPipeline method*), 1804
can_tune_threshold_with_objective() (*evalml.pipelines.TimeSeriesMulticlassClassificationPipeline method*), 1817
can_tune_threshold_with_objective() (*evalml.pipelines.TimeSeriesRegressionPipeline method*), 1825
cancel() (*evalml.automl.engine.cf_engine.CFComputation method*), 239
cancel() (*evalml.automl.engine.dask_engine.DaskComputation method*), 242
cancel() (*evalml.automl.engine.engine_base.EngineComputation method*), 245
cancel() (*evalml.automl.engine.EngineComputation method*), 254
cancel() (*evalml.automl.engine.sequential_engine.SequentialComputation method*), 248
CatBoostClassifier (*class in evalml.pipelines*), 1652
CatBoostClassifier (*class in evalml.pipelines.components*), 1346
CatBoostClassifier (*class in evalml.pipelines.components.estimators*), 868
CatBoostClassifier (*class in evalml.pipelines.components.estimators.classifiers*), 699
CatBoostClassifier (*class in evalml.pipelines.components.estimators.classifiers.catboost_classifier*), 647
CatBoostRegressor (*class in evalml.pipelines*), 1655
CatBoostRegressor (*class in evalml.pipelines.components*), 1350
CatBoostRegressor (*class in evalml.pipelines.components.estimators*), 872
CatBoostRegressor (*class in evalml.pipelines.components.estimators.regressors*), 808
CatBoostRegressor (*class in evalml.pipelines.components.estimators.regressors.catboost_regressor*), 750
categories() (*evalml.pipelines.components.OneHotEncoder method*), 1438
categories() (*evalml.pipelines.components.OrdinalEncoder method*), 1441
categories() (*evalml.pipelines.components.transformers.encoders.onehot_encoder.OneHotEncoder method*), 973
categories() (*evalml.pipelines.components.transformers.encoders.OneHotEncoder method*), 988

categories() (*evalml.pipelines.components.transformers.encoders.ordinal_encoder.OrdinalEncoder method*), 977
categories() (*evalml.pipelines.components.transformers.encoders.OrdinalEncoder method*), 991
categories() (*evalml.pipelines.components.transformers.OneHotEncoder method*), 1252
categories() (*evalml.pipelines.components.transformers.OrdinalEncoder method*), 1255
categories() (*evalml.pipelines.OneHotEncoder method*), 1734
categories() (*evalml.pipelines.OrdinalEncoder method*), 1737
CFClient (*class in evalml.automl.engine.cf_engine*), 239
CFComputation (*class in evalml.automl.engine.cf_engine*), 239
CFEngine (*class in evalml.automl.engine*), 250
CFEngine (*class in evalml.automl.engine.cf_engine*), 240
check_all_pipeline_names_unique() (*in module evalml.automl.utils*), 270
check_for_fit() (*evalml.pipelines.components.component_base.ComponentBase class method*), 1327
check_for_fit() (*evalml.pipelines.components.ComponentBase class method*), 1356
check_for_fit() (*evalml.pipelines.components.transformer.Transformer class method*), 975
check_for_fit() (*evalml.pipelines.components.transformer.Transformer class method*), 980
check_for_fit() (*evalml.pipelines.components.transformer.Transformer class method*), 1048
check_for_fit() (*evalml.pipelines.pipeline_meta.PipelineBaseMeta class method*), 1588
classes_ (*evalml.pipelines.binary_classification_pipeline.BinaryClassificationPipeline property*), 1556
classes_ (*evalml.pipelines.BinaryClassificationPipeline property*), 1648
classes_ (*evalml.pipelines.classification_pipeline.ClassificationPipeline property*), 1564
classes_ (*evalml.pipelines.ClassificationPipeline property*), 1661
classes_ (*evalml.pipelines.components.BaselineClassifier property*), 1341
classes_ (*evalml.pipelines.components.estimators.BaselineClassifier property*), 863
classes_ (*evalml.pipelines.components.estimators.classifier.BaselineClassifier property*), 644
classes_ (*evalml.pipelines.components.estimators.classifier.Classifier property*), 697
classes_ (*evalml.pipelines.multiclass_classification_pipeline.MulticlassClassificationPipeline property*), 1577
classes_ (*evalml.pipelines.MulticlassClassificationPipeline property*), 1729
classes_ (*evalml.pipelines.time_series_classification_pipeline.TimeSeriesBinaryClassificationPipeline property*), 1598
classes_ (*evalml.pipelines.time_series_classification_pipelines.TimeSeriesClassificationPipeline property*), 1606
classes_ (*evalml.pipelines.time_series_classification_pipelines.TimeSeriesMulticlassClassificationPipeline property*), 1614
classes_ (*evalml.pipelines.TimeSeriesBinaryClassificationPipeline property*), 1796
classes_ (*evalml.pipelines.TimeSeriesClassificationPipeline property*), 1804
classes_ (*evalml.pipelines.TimeSeriesMulticlassClassificationPipeline property*), 1817
ClassificationPipeline (*class in evalml.pipelines*), 1658
ClassificationPipeline (*class in evalml.pipelines.classification_pipeline*), 1562
ClassImbalanceDataCheck (*class in evalml.data_checks*), 343
ClassImbalanceDataCheck (*class in evalml.data_checks.class_imbalance_data_check*), 286
classproperty (*class in evalml.utils*), 1910
classproperty (*class in evalml.utils.gen_utils*), 1902
clone() (*evalml.pipelines.ARIMARegressor method*), 1643
clone() (*evalml.pipelines.binary_classification_pipeline.BinaryClassificationPipeline method*), 1556
clone() (*evalml.pipelines.BinaryClassificationPipeline method*), 1648
clone() (*evalml.pipelines.CatBoostClassifier method*), 1653
clone() (*evalml.pipelines.CatBoostRegressor method*), 1656
clone() (*evalml.pipelines.classification_pipeline.ClassificationPipeline method*), 1564
clone() (*evalml.pipelines.ClassificationPipeline method*), 1661
clone() (*evalml.pipelines.components.ARIMARegressor method*), 1338
clone() (*evalml.pipelines.components.BaselineClassifier method*), 1341
clone() (*evalml.pipelines.components.BaselineRegressor method*), 1344
clone() (*evalml.pipelines.components.CatBoostClassifier method*), 1348
clone() (*evalml.pipelines.components.CatBoostRegressor method*), 1351
clone() (*evalml.pipelines.components.component_base.ComponentBase method*), 1324
clone() (*evalml.pipelines.components.ComponentBase method*), 1354
clone() (*evalml.pipelines.components.DateTimeFeaturizer method*), 1357
clone() (*evalml.pipelines.components.DecisionTreeClassifier method*), 1360
clone() (*evalml.pipelines.components.DecisionTreeRegressor*

*method*), 1364
clone() (*evalml.pipelines.components.DFSTransformer method*), 1367
clone() (*evalml.pipelines.components.DropColumns method*), 1370
clone() (*evalml.pipelines.components.DropNaNRowsTransformer method*), 1372
clone() (*evalml.pipelines.components.DropNullColumns method*), 1375
clone() (*evalml.pipelines.components.DropRowsTransformer method*), 1377
clone() (*evalml.pipelines.components.ElasticNetClassifier method*), 1380
clone() (*evalml.pipelines.components.ElasticNetRegressor method*), 1383
clone() (*evalml.pipelines.components.EmailFeaturizer method*), 1386
clone() (*evalml.pipelines.components.ensemble.stacked_ensemble_base.StackedEnsembleBase method*), 620
clone() (*evalml.pipelines.components.ensemble.stacked_ensemble_classifier.StackedEnsembleClassifier method*), 624
clone() (*evalml.pipelines.components.ensemble.stacked_ensemble_regressor.StackedEnsembleRegressor method*), 628
clone() (*evalml.pipelines.components.ensemble.StackedEnsembleBase method*), 632
clone() (*evalml.pipelines.components.ensemble.StackedEnsembleClassifier method*), 636
clone() (*evalml.pipelines.components.ensemble.StackedEnsembleRegressor method*), 640
clone() (*evalml.pipelines.components.Estimator method*), 1389
clone() (*evalml.pipelines.components.estimators.ARIMARegressor method*), 860
clone() (*evalml.pipelines.components.estimators.BaselineClassifier method*), 863
clone() (*evalml.pipelines.components.estimators.BaselineRegressor method*), 866
clone() (*evalml.pipelines.components.estimators.CatBoostClassifier method*), 870
clone() (*evalml.pipelines.components.estimators.CatBoostRegressor method*), 873
clone() (*evalml.pipelines.components.estimators.classifiers.baseline_classifier.BaselineClassifier method*), 644
clone() (*evalml.pipelines.components.estimators.classifiers.catboost_classifier.CatBoostClassifier method*), 697
clone() (*evalml.pipelines.components.estimators.classifiers.cb_classifier.CatBoostClassifier method*), 648
clone() (*evalml.pipelines.components.estimators.classifiers.catboost_classifier.CatBoostClassifier method*), 700
clone() (*evalml.pipelines.components.estimators.classifiers.decision_tree_classifier.DecisionTreeClassifier method*), 652
clone() (*evalml.pipelines.components.estimators.classifiers.rf_classifier.RandomForestClassifier method*), 704
clone() (*evalml.pipelines.components.estimators.classifiers.et_classifier.ExtraTreesClassifier method*), 656
clone() (*evalml.pipelines.components.estimators.classifiers.ElasticNetClassifier method*), 707
clone() (*evalml.pipelines.components.estimators.classifiers.et_classifier.ExtraTreesClassifier method*), 660
clone() (*evalml.pipelines.components.estimators.classifiers.ExtraTreesClassifier method*), 711
clone() (*evalml.pipelines.components.estimators.classifiers.kneighbors_classifier.KNeighborsClassifier method*), 664
clone() (*evalml.pipelines.components.estimators.classifiers.KNeighborsClassifier method*), 715
clone() (*evalml.pipelines.components.estimators.classifiers.lightgbm_classifier.LightGBMClassifier method*), 668
clone() (*evalml.pipelines.components.estimators.classifiers.LightGBMClassifier method*), 719
clone() (*evalml.pipelines.components.estimators.classifiers.logistic_regression_classifier.LogisticRegressionClassifier method*), 672
clone() (*evalml.pipelines.components.estimators.classifiers.LogisticRegressionClassifier method*), 722
clone() (*evalml.pipelines.components.estimators.classifiers.rf_classifier.RandomForestClassifier method*), 725
clone() (*evalml.pipelines.components.estimators.classifiers.rf_classifier.RandomForestClassifier method*), 676
clone() (*evalml.pipelines.components.estimators.classifiers.svm_classifier.SVMClassifier method*), 680
clone() (*evalml.pipelines.components.estimators.classifiers.SVMClassifier method*), 728
clone() (*evalml.pipelines.components.estimators.classifiers.vowpal_wabbit_base_classifier.VowpalWabbitBaseClassifier method*), 683
clone() (*evalml.pipelines.components.estimators.classifiers.vowpal_wabbit_base_classifier.VowpalWabbitBaseClassifier method*), 686
clone() (*evalml.pipelines.components.estimators.classifiers.vowpal_wabbit_multiclass_classifier.VowpalWabbitMulticlassClassifier method*), 690
clone() (*evalml.pipelines.components.estimators.classifiers.VowpalWabbitBaseClassifier method*), 732
clone() (*evalml.pipelines.components.estimators.classifiers.VowpalWabbitMulticlassClassifier method*), 735
clone() (*evalml.pipelines.components.estimators.classifiers.xgboost_classifier.XGBoostClassifier method*), 693
clone() (*evalml.pipelines.components.estimators.classifiers.XGBoostClassifier method*), 738
clone() (*evalml.pipelines.components.estimators.DecisionTreeClassifier method*), 877
clone() (*evalml.pipelines.components.estimators.DecisionTreeRegressor method*), 881
clone() (*evalml.pipelines.components.estimators.ElasticNetClassifier method*), 884
clone() (*evalml.pipelines.components.estimators.ElasticNetRegressor method*), 887
clone() (*evalml.pipelines.components.estimators.Estimator method*), 890
clone() (*evalml.pipelines.components.estimator.Estimator method*), 855
clone() (*evalml.pipelines.components.estimators.ExponentialSmoothingRe*

*method*), 894
clone() (*evalml.pipelines.components.estimators.ExtraTreesClassifier method*), 898
clone() (*evalml.pipelines.components.estimators.ExtraTreesRegressor method*), 902
clone() (*evalml.pipelines.components.estimators.KNeighborsClassifier method*), 905
clone() (*evalml.pipelines.components.estimators.LightGBMClassifier method*), 909
clone() (*evalml.pipelines.components.estimators.LightGBMRegressor method*), 913
clone() (*evalml.pipelines.components.estimators.LinearRegressor method*), 916
clone() (*evalml.pipelines.components.estimators.LogisticRegressionClassifier method*), 919
clone() (*evalml.pipelines.components.estimators.ProphetRegressor method*), 923
clone() (*evalml.pipelines.components.estimators.RandomForestClassifier method*), 926
clone() (*evalml.pipelines.components.estimators.RandomForestRegressor method*), 929
clone() (*evalml.pipelines.components.estimators.regressors.arima_regressor method*), 744
clone() (*evalml.pipelines.components.estimators.regressors.ARIMARegressor method*), 803
clone() (*evalml.pipelines.components.estimators.regressors.baseline_regressor method*), 747
clone() (*evalml.pipelines.components.estimators.regressors.BaselineRegressor method*), 806
clone() (*evalml.pipelines.components.estimators.regressors.catboost_regressor method*), 751
clone() (*evalml.pipelines.components.estimators.regressors.CatBoostRegressor method*), 810
clone() (*evalml.pipelines.components.estimators.regressors.decision_tree_regressor method*), 755
clone() (*evalml.pipelines.components.estimators.regressors.DecisionTreeRegressor method*), 813
clone() (*evalml.pipelines.components.estimators.regressors.elastic_net_regressor method*), 759
clone() (*evalml.pipelines.components.estimators.regressors.ElasticNetRegressor method*), 817
clone() (*evalml.pipelines.components.estimators.regressors.extra_trees_regressor method*), 763
clone() (*evalml.pipelines.components.estimators.regressors.ExtraTreesRegressor method*), 768
clone() (*evalml.pipelines.components.estimators.regressors.exponential_smoothing_regressor method*), 821
clone() (*evalml.pipelines.components.estimators.regressors.ExponentialSmoothingRegressor method*), 824
clone() (*evalml.pipelines.components.estimators.regressors.linear_regressor method*), 771
clone() (*evalml.pipelines.components.estimators.regressors.LinearRegressor method*), 828
clone() (*evalml.pipelines.components.estimators.regressors.prophet_regressor method*), 775
clone() (*evalml.pipelines.components.estimators.regressors.ProphetRegressor method*), 831
clone() (*evalml.pipelines.components.estimators.regressors.rf_regressor method*), 835
clone() (*evalml.pipelines.components.estimators.regressors.RandomForestRegressor method*), 838
clone() (*evalml.pipelines.components.estimators.regressors.rf_regressor.RandomForestRegressor method*), 784
clone() (*evalml.pipelines.components.estimators.regressors.svm_regressor method*), 787
clone() (*evalml.pipelines.components.estimators.regressors.SVMRegressor method*), 841
clone() (*evalml.pipelines.components.estimators.regressors.time_series_baseline_regressor method*), 791
clone() (*evalml.pipelines.components.estimators.regressors.TimeSeriesBaselineRegressor method*), 844
clone() (*evalml.pipelines.components.estimators.regressors.vowpal_wabbit_regressor method*), 794
clone() (*evalml.pipelines.components.estimators.regressors.VowpalWabbitRegressor method*), 848
clone() (*evalml.pipelines.components.estimators.regressors.xgboost_regressor method*), 798
clone() (*evalml.pipelines.components.estimators.regressors.XGBoostRegressor method*), 851
clone() (*evalml.pipelines.components.estimators.SVMClassifier method*), 932
clone() (*evalml.pipelines.components.estimators.SVMRegressor method*), 935
clone() (*evalml.pipelines.components.estimators.TimeSeriesBaselineEstimator method*), 938
clone() (*evalml.pipelines.components.estimators.VowpalWabbitBinaryClassifier method*), 942
clone() (*evalml.pipelines.components.estimators.VowpalWabbitMulticlassClassifier method*), 945
clone() (*evalml.pipelines.components.estimators.VowpalWabbitRegressor method*), 948
clone() (*evalml.pipelines.components.estimators.XGBoostClassifier method*), 951
clone() (*evalml.pipelines.components.estimators.XGBoostRegressor method*), 954
clone() (*evalml.pipelines.components.ExponentialSmoothingRegressor method*), 1392
clone() (*evalml.pipelines.components.ExtraTreesClassifier method*), 1396
clone() (*evalml.pipelines.components.ExtraTreesRegressor method*), 1399
clone() (*evalml.pipelines.components.LightGBMRegressorFeatureSelector method*), 1402
clone() (*evalml.pipelines.components.Imputer method*), 1405
clone() (*evalml.pipelines.components.KNeighborsClassifier*

*method*), 1408
clone() (*evalml.pipelines.components.LabelEncoder method*), 1411
clone() (*evalml.pipelines.components.LightGBMClassifier method*), 1414
clone() (*evalml.pipelines.components.LightGBMRegressor method*), 1418
clone() (*evalml.pipelines.components.LinearDiscriminantAnalysis method*), 1421
clone() (*evalml.pipelines.components.LinearRegressor method*), 1423
clone() (*evalml.pipelines.components.LogisticRegressionClassifier method*), 1427
clone() (*evalml.pipelines.components.LogTransformer method*), 1430
clone() (*evalml.pipelines.components.LSA method*), 1432
clone() (*evalml.pipelines.components.NaturalLanguageFeaturizer method*), 1435
clone() (*evalml.pipelines.components.OneHotEncoder method*), 1438
clone() (*evalml.pipelines.components.OrdinalEncoder method*), 1441
clone() (*evalml.pipelines.components.Oversampler method*), 1444
clone() (*evalml.pipelines.components.PCA method*), 1446
clone() (*evalml.pipelines.components.PerColumnImputer method*), 1449
clone() (*evalml.pipelines.components.PolynomialDecomposer method*), 1452
clone() (*evalml.pipelines.components.ProphetRegressor method*), 1457
clone() (*evalml.pipelines.components.RandomForestClassifier method*), 1460
clone() (*evalml.pipelines.components.RandomForestRegressor method*), 1463
clone() (*evalml.pipelines.components.ReplaceNullableTypes method*), 1466
clone() (*evalml.pipelines.components.RFClassifierRFESelector method*), 1469
clone() (*evalml.pipelines.components.RFClassifierSelectFromModel method*), 1472
clone() (*evalml.pipelines.components.RFRegressorRFESelector method*), 1475
clone() (*evalml.pipelines.components.RFRegressorSelectFromModel method*), 1478
clone() (*evalml.pipelines.components.SelectByType method*), 1480
clone() (*evalml.pipelines.components.SelectColumns method*), 1483
clone() (*evalml.pipelines.components.SimpleImputer method*), 1485
clone() (*evalml.pipelines.components.StackedEnsembleBase method*), 1488
clone() (*evalml.pipelines.components.StackedEnsembleClassifier method*), 1492
clone() (*evalml.pipelines.components.StackedEnsembleRegressor method*), 1496
clone() (*evalml.pipelines.components.StandardScaler method*), 1498
clone() (*evalml.pipelines.components.STLDecomposer method*), 1501
clone() (*evalml.pipelines.components.SVMClassifier method*), 1506
clone() (*evalml.pipelines.components.SVMRegressor method*), 1509
clone() (*evalml.pipelines.components.TargetEncoder method*), 1512
clone() (*evalml.pipelines.components.TargetImputer method*), 1515
clone() (*evalml.pipelines.components.TimeSeriesBaselineEstimator method*), 1518
clone() (*evalml.pipelines.components.TimeSeriesFeaturizer method*), 1521
clone() (*evalml.pipelines.components.TimeSeriesImputer method*), 1524
clone() (*evalml.pipelines.components.TimeSeriesRegularizer method*), 1527
clone() (*evalml.pipelines.components.Transformer method*), 1530
clone() (*evalml.pipelines.components.transformers.column_selectors.Colu method*), 1202
clone() (*evalml.pipelines.components.transformers.column_selectors.Drop method*), 1204
clone() (*evalml.pipelines.components.transformers.column_selectors.Sele method*), 1207
clone() (*evalml.pipelines.components.transformers.column_selectors.Sele method*), 1209
clone() (*evalml.pipelines.components.transformers.DateTimeFeaturizer method*), 1216
clone() (*evalml.pipelines.components.transformers.DFSTransformer method*), 1219
clone() (*evalml.pipelines.components.transformers.dimensionality_reduct method*), 958
clone() (*evalml.pipelines.components.transformers.dimensionality_reduct method*), 964
clone() (*evalml.pipelines.components.transformers.dimensionality_reduct method*), 966
clone() (*evalml.pipelines.components.transformers.dimensionality_reduct method*), 961
clone() (*evalml.pipelines.components.transformers.DropColumns method*), 1222
clone() (*evalml.pipelines.components.transformers.DropNaNRowsTransfo method*), 1224
clone() (*evalml.pipelines.components.transformers.DropNullColumns method*), 1226
clone() (*evalml.pipelines.components.transformers.DropRowsTransformer

*method*), 1229
clone() (*evalml.pipelines.components.transformers.EmailFeature*
*method*), 1231
clone() (*evalml.pipelines.components.transformers.encoders.LabelEncoder*
*method*), 969
clone() (*evalml.pipelines.components.transformers.encoders.LabelEncoder*
*method*), 985
clone() (*evalml.pipelines.components.transformers.encoders.OneHotEncoder*
*method*), 973
clone() (*evalml.pipelines.components.transformers.encoders.OneHotEncoder*
*method*), 988
clone() (*evalml.pipelines.components.transformers.encoders.OrdinalEncoder*
*method*), 978
clone() (*evalml.pipelines.components.transformers.encoders.OrdinalEncoder*
*method*), 992
clone() (*evalml.pipelines.components.transformers.encoders.TargetEncoder*
*method*), 982
clone() (*evalml.pipelines.components.transformers.encoders.TargetEncoder*
*method*), 994
clone() (*evalml.pipelines.components.transformers.feature_selection.ComponentSelector*
*method*), 997
clone() (*evalml.pipelines.components.transformers.feature_selection.FeatureSelector*
*method*), 1017
clone() (*evalml.pipelines.components.transformers.feature_selection.RecursiveFeatureElimination*
*method*), 1001
clone() (*evalml.pipelines.components.transformers.feature_selection.RFCheckifEnRFES*
*method*), 1004
clone() (*evalml.pipelines.components.transformers.feature_selection.RFRegressorRFES*
*method*), 1007
clone() (*evalml.pipelines.components.transformers.feature_selection.RFClassifierSelectFromModel*
*method*), 1011
clone() (*evalml.pipelines.components.transformers.feature_selection.RFRegressorSelectFromModel*
*method*), 1014
clone() (*evalml.pipelines.components.transformers.feature_selection.RFClassifierRFESelector*
*method*), 1021
clone() (*evalml.pipelines.components.transformers.feature_selection.RFClassifierSelectFromModel*
*method*), 1024
clone() (*evalml.pipelines.components.transformers.feature_selection.RFRegressorRFESelector*
*method*), 1027
clone() (*evalml.pipelines.components.transformers.feature_selection.RFRegressorSelectFromModel*
*method*), 1030
clone() (*evalml.pipelines.components.transformers.FeatureSelector*
*method*), 1234
clone() (*evalml.pipelines.components.transformers.Imputer*
*method*), 1237
clone() (*evalml.pipelines.components.transformers.imputers.BaseImputer*
*method*), 1053
clone() (*evalml.pipelines.components.transformers.imputers.BaseImputer*
*method*), 1034
clone() (*evalml.pipelines.components.transformers.imputers.KNNImputer*
*method*), 1037
clone() (*evalml.pipelines.components.transformers.imputers.KNNImputer*
*method*), 1055
clone() (*evalml.pipelines.components.transformers.imputers.PerColumnImputer*
*method*), 1040
clone() (*evalml.pipelines.components.transformers.imputers.PerColumnImputer*
*method*), 1058
clone() (*evalml.pipelines.components.transformers.imputers.simple_imputer.SimpleImputer*
*method*), 1043
clone() (*evalml.pipelines.components.transformers.imputers.SimpleImputer*
*method*), 1060
clone() (*evalml.pipelines.components.transformers.imputers.target_imputer.TargetImputer*
*method*), 1046
clone() (*evalml.pipelines.components.transformers.imputers.TargetImputer*
*method*), 1063
clone() (*evalml.pipelines.components.transformers.imputers.time_series_imputer.TimeSeriesImputer*
*method*), 1050
clone() (*evalml.pipelines.components.transformers.imputers.TimeSeriesImputer*
*method*), 1066
clone() (*evalml.pipelines.components.transformers.LabelEncoder*
*method*), 1239
clone() (*evalml.pipelines.components.transformers.LinearDiscriminantAnalysis*
*method*), 1242
clone() (*evalml.pipelines.components.transformers.LogTransformer*
*method*), 1244
clone() (*evalml.pipelines.components.transformers.LSA*
*method*), 1247
clone() (*evalml.pipelines.components.transformers.RemoveNullRows*
*method*), 1249
clone() (*evalml.pipelines.components.transformers.RFCheckifEnRFES*
*method*), 1252
clone() (*evalml.pipelines.components.transformers.RFRegressorRFES*
*method*), 1256
clone() (*evalml.pipelines.components.transformers.RFClassifierSelectFromModel*
*method*), 1259
clone() (*evalml.pipelines.components.transformers.RFRegressorSelectFromModel*
*method*), 1261
clone() (*evalml.pipelines.components.transformers.PerColumnImputer*
*method*), 1264
clone() (*evalml.pipelines.components.transformers.PolynomialDecomposer*
*method*), 1267
clone() (*evalml.pipelines.components.transformers.preprocessing.datetime.*
*method*), 1069
clone() (*evalml.pipelines.components.transformers.preprocessing.DateTime*
*method*), 1129
clone() (*evalml.pipelines.components.transformers.preprocessing.Decomp*
*method*), 1132
clone() (*evalml.pipelines.components.transformers.preprocessing.decomp*
*method*), 1073
clone() (*evalml.pipelines.components.transformers.preprocessing.DFSTra*
*method*), 1136
clone() (*evalml.pipelines.components.transformers.preprocessing.drop_na*
*method*), 1077
clone() (*evalml.pipelines.components.transformers.preprocessing.drop_nu*
*method*), 1080
clone() (*evalml.pipelines.components.transformers.preprocessing.drop_ro*
*method*), 1083
clone() (*evalml.pipelines.components.transformers.preprocessing.DropNa*

*method*), 1139
clone() (*evalml.pipelines.components.transformers.preprocessing.DateTime*
*method*), 1141
clone() (*evalml.pipelines.components.transformers.preprocessing.DropNullColumns*
*method*), 1144
clone() (*evalml.pipelines.components.transformers.preprocessing.DropRowsTransformer*
*method*), 1146
clone() (*evalml.pipelines.components.transformers.preprocessing.DFSTransformer*
*method*), 1086
clone() (*evalml.pipelines.components.transformers.preprocessing.LogTransformer*
*method*), 1089
clone() (*evalml.pipelines.components.transformers.preprocessing.LogTransformer*
*method*), 1148
clone() (*evalml.pipelines.components.transformers.preprocessing.LSA*
*method*), 1151
clone() (*evalml.pipelines.components.transformers.preprocessing.LSA*
*method*), 1092
clone() (*evalml.pipelines.components.transformers.preprocessing.NaturalLanguageFeaturizer*
*method*), 1095
clone() (*evalml.pipelines.components.transformers.preprocessing.NaturalLanguageFeaturizer*
*method*), 1153
clone() (*evalml.pipelines.components.transformers.preprocessing.PolynomialDecomposer*
*method*), 1099
clone() (*evalml.pipelines.components.transformers.preprocessing.PolynomialDecomposer*
*method*), 1157
clone() (*evalml.pipelines.components.transformers.preprocessing.ReplaceNullableTypes*
*method*), 1104
clone() (*evalml.pipelines.components.transformers.preprocessing.ReplaceNullableTypes*
*method*), 1161
clone() (*evalml.pipelines.components.transformers.preprocessing.STLDecomposer*
*method*), 1108
clone() (*evalml.pipelines.components.transformers.preprocessing.STLDecomposer*
*method*), 1164
clone() (*evalml.pipelines.components.transformers.preprocessing.TextFeaturizer*
*method*), 1113
clone() (*evalml.pipelines.components.transformers.preprocessing.TextFeaturizer*
*method*), 1169
clone() (*evalml.pipelines.components.transformers.preprocessing.TimeSeriesFeaturizer*
*method*), 1117
clone() (*evalml.pipelines.components.transformers.preprocessing.TimeSeriesFeaturizer*
*method*), 1120
clone() (*evalml.pipelines.components.transformers.preprocessing.TimeSeriesRegularizer*
*method*), 1172
clone() (*evalml.pipelines.components.transformers.preprocessing.TimeSeriesRegularizer*
*method*), 1175
clone() (*evalml.pipelines.components.transformers.preprocessing.TransformPrimitiveComponent*
*method*), 1123
clone() (*evalml.pipelines.components.transformers.preprocessing.UrlFeaturizer*
*method*), 1126
clone() (*evalml.pipelines.components.transformers.preprocessing.UrlFeaturizer*
*method*), 1178
clone() (*evalml.pipelines.components.transformers.ReplaceNullableTypes*
*method*), 1271
clone() (*evalml.pipelines.components.transformers.RFClassifierRFESelect*
*method*), 1274
clone() (*evalml.pipelines.components.transformers.RFClassifierSelectFro*
*method*), 1277
clone() (*evalml.pipelines.components.transformers.RFRegressorRFESelec*
*method*), 1280
clone() (*evalml.pipelines.components.transformers.RFRegressorSelectFro*
*method*), 1283
clone() (*evalml.pipelines.components.transformers.samplers.base_sample*
*method*), 1181
clone() (*evalml.pipelines.components.transformers.samplers.Oversampler*
*method*), 1190
clone() (*evalml.pipelines.components.transformers.samplers.oversampler.*
*method*), 1184
clone() (*evalml.pipelines.components.transformers.samplers.Undersampl*
*method*), 1193
clone() (*evalml.pipelines.components.transformers.samplers.undersample*
*method*), 1187
clone() (*evalml.pipelines.components.transformers.scalers.standard_scale*
*method*), 1196
clone() (*evalml.pipelines.components.transformers.scalers.StandardScale*
*method*), 1199
clone() (*evalml.pipelines.components.transformer.SelectByType*
*method*), 1286
clone() (*evalml.pipelines.components.transformers.SelectColumns*
*method*), 1288
clone() (*evalml.pipelines.components.transformers.SimpleImputer*
*method*), 1290
clone() (*evalml.pipelines.components.transformers.StandardScaler*
*method*), 1293
clone() (*evalml.pipelines.components.transformers.STLDecomposer*
*method*), 1296
clone() (*evalml.pipelines.components.transformers.TargetEncoder*
*method*), 1301
clone() (*evalml.pipelines.components.transformers.TargetImputer*
*method*), 1304
clone() (*evalml.pipelines.components.transformers.TimeSeriesFeaturizer*
*method*), 1307
clone() (*evalml.pipelines.components.transformers.TimeSeriesImputer*
*method*), 1310
clone() (*evalml.pipelines.components.transformers.TimeSeriesRegularizer*
*method*), 1313
clone() (*evalml.pipelines.components.transformers.Transformer*
*method*), 1316
clone() (*evalml.pipelines.components.transformers.transformer.Transform*
*method*), 1212
clone() (*evalml.pipelines.components.transformer_primitive_components.EmptyFeatureUndersampler*
*method*), 1319
clone() (*evalml.pipelines.components.transformer_primitive_components.UrlFeaturizer*
*method*), 1321
clone() (*evalml.pipelines.components.Undersampler*
*method*), 1533
clone() (*evalml.pipelines.components.UrlFeaturizer*
*method*), 1535
clone() (*evalml.pipelines.components.VowpalWabbitBinaryClassifier*

*method*), 1538
clone() (*evalml.pipelines.components.VowpalWabbitMulticlassClassifier method*), 1541
clone() (*evalml.pipelines.components.VowpalWabbitRegressor method*), 1544
clone() (*evalml.pipelines.components.XGBoostClassifier method*), 1547
clone() (*evalml.pipelines.components.XGBoostRegressor method*), 1550
clone() (*evalml.pipelines.DecisionTreeClassifier method*), 1672
clone() (*evalml.pipelines.DecisionTreeRegressor method*), 1676
clone() (*evalml.pipelines.DFSTransformer method*), 1679
clone() (*evalml.pipelines.DropNaNRowsTransformer method*), 1682
clone() (*evalml.pipelines.ElasticNetClassifier method*), 1685
clone() (*evalml.pipelines.ElasticNetRegressor method*), 1688
clone() (*evalml.pipelines.Estimator method*), 1691
clone() (*evalml.pipelines.ExponentialSmoothingRegressor method*), 1694
clone() (*evalml.pipelines.ExtraTreesClassifier method*), 1698
clone() (*evalml.pipelines.ExtraTreesRegressor method*), 1702
clone() (*evalml.pipelines.FeatureSelector method*), 1705
clone() (*evalml.pipelines.Imputer method*), 1708
clone() (*evalml.pipelines.KNeighborsClassifier method*), 1711
clone() (*evalml.pipelines.LightGBMClassifier method*), 1714
clone() (*evalml.pipelines.LightGBMRegressor method*), 1718
clone() (*evalml.pipelines.LinearRegressor method*), 1721
clone() (*evalml.pipelines.LogisticRegressionClassifier method*), 1724
clone() (*evalml.pipelines.multiclass_classification_pipeline.MulticlassClassificationPipeline method*), 1577
clone() (*evalml.pipelines.MulticlassClassificationPipeline method*), 1729
clone() (*evalml.pipelines.OneHotEncoder method*), 1734
clone() (*evalml.pipelines.OrdinalEncoder method*), 1738
clone() (*evalml.pipelines.PerColumnImputer method*), 1740
clone() (*evalml.pipelines.pipeline_base.PipelineBase method*), 1583
clone() (*evalml.pipelines.PipelineBase method*), 1743
clone() (*evalml.pipelines.ProphetRegressor method*), 1749
clone() (*evalml.pipelines.RandomForestClassifier method*), 1752
clone() (*evalml.pipelines.RandomForestRegressor method*), 1755
clone() (*evalml.pipelines.regression_pipeline.RegressionPipeline method*), 1591
clone() (*evalml.pipelines.RegressionPipeline method*), 1759
clone() (*evalml.pipelines.RFClassifierSelectFromModel method*), 1764
clone() (*evalml.pipelines.RFRegressorSelectFromModel method*), 1767
clone() (*evalml.pipelines.SimpleImputer method*), 1770
clone() (*evalml.pipelines.StackedEnsembleBase method*), 1773
clone() (*evalml.pipelines.StackedEnsembleClassifier method*), 1777
clone() (*evalml.pipelines.StackedEnsembleRegressor method*), 1781
clone() (*evalml.pipelines.StandardScaler method*), 1783
clone() (*evalml.pipelines.SVMClassifier method*), 1786
clone() (*evalml.pipelines.SVMRegressor method*), 1789
clone() (*evalml.pipelines.TargetEncoder method*), 1792
clone() (*evalml.pipelines.time_series_classification_pipelines.TimeSeries method*), 1598
clone() (*evalml.pipelines.time_series_classification_pipelines.TimeSeries method*), 1606
clone() (*evalml.pipelines.time_series_classification_pipelines.TimeSeries method*), 1614
clone() (*evalml.pipelines.time_series_pipeline_base.TimeSeriesPipelineB method*), 1622
clone() (*evalml.pipelines.time_series_regression_pipeline.TimeSeriesReg method*), 1629
clone() (*evalml.pipelines.TimeSeriesBinaryClassificationPipeline method*), 1796
clone() (*evalml.pipelines.TimeSeriesClassificationPipeline method*), 1804
clone() (*evalml.pipelines.TimeSeriesFeaturizer method*), 1810
clone() (*evalml.pipelines.TimeSeriesImputer method*), 1813
clone() (*evalml.pipelines.TimeSeriesMulticlassClassificationPipeline method*), 1817
clone() (*evalml.pipelines.TimeSeriesRegressionPipeline method*), 1825
clone() (*evalml.pipelines.TimeSeriesRegularizer method*), 1832
clone() (*evalml.pipelines.Transformer method*), 1835
clone() (*evalml.pipelines.VowpalWabbitBinaryClassifier method*), 1838
clone() (*evalml.pipelines.VowpalWabbitMulticlassClassifier

*method*), 1841
clone() (*evalml.pipelines.VowpalWabbitRegressor method*), 1844
clone() (*evalml.pipelines.XGBoostClassifier method*), 1847
clone() (*evalml.pipelines.XGBoostRegressor method*), 1850
close() (*evalml.automl.engine.cf_engine.CFClient method*), 239
close() (*evalml.automl.engine.cf_engine.CFEngine method*), 240
close() (*evalml.automl.engine.CFEngine method*), 251
close() (*evalml.automl.engine.dask_engine.DaskEngine method*), 242
close() (*evalml.automl.engine.DaskEngine method*), 252
close() (*evalml.automl.engine.sequential_engine.SequentialEngine method*), 249
close() (*evalml.automl.engine.SequentialEngine method*), 255
close() (*evalml.automl.SequentialEngine method*), 284
close_engine() (*evalml.automl.automl_search.AutoMLSearch method*), 261
close_engine() (*evalml.automl.AutoMLSearch method*), 277
close_engine() (*evalml.AutoMLSearch method*), 1917
ColumnSelector (*class in evalml.pipelines.components.transformers.column_selectors*), 1201
ComponentBase (*class in evalml.pipelines.components*), 1353
ComponentBase (*class in evalml.pipelines.components.component_base*), 1324
ComponentBaseMeta (*class in evalml.pipelines.components*), 1355
ComponentBaseMeta (*class in evalml.pipelines.components.component_base_meta*), 1326
ComponentGraph (*class in evalml.pipelines*), 1665
ComponentGraph (*class in evalml.pipelines.component_graph*), 1568
ComponentNotYetFittedError, 392, 395
compute_order (*evalml.pipelines.component_graph.ComponentGraph property*), 1570
compute_order (*evalml.pipelines.ComponentGraph property*), 1667
CONDA_TO_PIP_NAME (*in module evalml.utils.cli_utils*), 1899
confusion_matrix() (*in module evalml.model_understanding*), 429
confusion_matrix() (*in module evalml.model_understanding.metrics*), 413
contains_all_ts_parameters() (*in module evalml.utils.gen_utils*), 1903
contains_pre_existing_features() (*evalml.pipelines.components.DFSTransformer static method*), 1367
contains_pre_existing_features() (*evalml.pipelines.components.transformers.DFSTransformer static method*), 1219
contains_pre_existing_features() (*evalml.pipelines.components.transformers.preprocessing.DFSTra static method*), 1136
contains_pre_existing_features() (*evalml.pipelines.components.transformers.preprocessing.feature static method*), 1086
contains_pre_existing_features() (*evalml.pipelines.DFSTransformer static method*), 1679
convert_dict_to_action() (*evalml.data_checks.data_check_action.DataCheckAction static method*), 290
convert_dict_to_action() (*evalml.data_checks.DataCheckAction static method*), 346
convert_dict_to_option() (*evalml.data_checks.data_check_action_option.DataCheckAction static method*), 292
convert_dict_to_option() (*evalml.data_checks.DataCheckActionOption static method*), 347
convert_to_seconds() (*in module evalml.utils*), 1910
convert_to_seconds() (*in module evalml.utils.gen_utils*), 1903
CostBenefitMatrix (*class in evalml.objectives*), 552
CostBenefitMatrix (*class in evalml.objectives.cost_benefit_matrix*), 444
create_objectives() (*evalml.pipelines.binary_classification_pipeline.BinaryClassifica static method*), 1556
create_objectives() (*evalml.pipelines.BinaryClassificationPipeline static method*), 1648
create_objectives() (*evalml.pipelines.classification_pipeline.ClassificationPipeline static method*), 1564
create_objectives() (*evalml.pipelines.ClassificationPipeline static method*), 1661
create_objectives() (*evalml.pipelines.multiclass_classification_pipeline.MulticlassCla static method*), 1577
create_objectives() (*evalml.pipelines.MulticlassClassificationPipeline static method*), 1729
create_objectives() (*evalml.pipelines.pipeline_base.PipelineBase

*static method*), 1584
create_objectives() (*evalml.pipelines.PipelineBase static method*), 1744
create_objectives() (*evalml.pipelines.regression_pipeline.RegressionPipeline static method*), 1592
create_objectives() (*evalml.pipelines.RegressionPipeline static method*), 1760
create_objectives() (*evalml.pipelines.time_series_classification_pipelines.TimeSeriesBinaryClassificationPipeline static method*), 1598
create_objectives() (*evalml.pipelines.time_series_classification_pipelines.TimeSeriesMulticlassClassificationPipeline static method*), 1606
create_objectives() (*evalml.pipelines.time_series_classification_pipelines.TimeSeriesClassificationPipeline static method*), 1614
create_objectives() (*evalml.pipelines.time_series_pipeline_base.TimeSeriesPipelineBase static method*), 1622
create_objectives() (*evalml.pipelines.time_series_regression_pipeline.TimeSeriesRegressionPipeline static method*), 1629
create_objectives() (*evalml.pipelines.TimeSeriesBinaryClassificationPipeline static method*), 1797
create_objectives() (*evalml.pipelines.TimeSeriesClassificationPipeline static method*), 1804
create_objectives() (*evalml.pipelines.TimeSeriesMulticlassClassificationPipeline static method*), 1817
create_objectives() (*evalml.pipelines.TimeSeriesRegressionPipeline static method*), 1825
cross_entropy() (*in module evalml.model_understanding.prediction_explanations.explainers*), 403
custom_name (*evalml.pipelines.binary_classification_pipeline.BinaryClassificationPipeline property*), 1556
custom_name (*evalml.pipelines.BinaryClassificationPipeline property*), 1648
custom_name (*evalml.pipelines.classification_pipeline.ClassificationPipeline property*), 1564
custom_name (*evalml.pipelines.ClassificationPipeline property*), 1661
custom_name (*evalml.pipelines.multiclass_classification_pipeline.MulticlassClassificationPipeline property*), 1577
custom_name (*evalml.pipelines.MulticlassClassificationPipeline property*), 1729
custom_name (*evalml.pipelines.pipeline_base.PipelineBase property*), 1584
custom_name (*evalml.pipelines.PipelineBase property*), 1744
custom_name (*evalml.pipelines.regression_pipeline.RegressionPipeline property*), 1592
custom_name (*evalml.pipelines.RegressionPipeline property*), 1760
custom_name (*evalml.pipelines.time_series_classification_pipelines.TimeSeriesBinaryClassificationPipeline property*), 1599
custom_name (*evalml.pipelines.time_series_classification_pipelines.TimeSeriesMulticlassClassificationPipeline property*), 1606
custom_name (*evalml.pipelines.time_series_classification_pipelines.TimeSeriesClassificationPipeline property*), 1614
custom_name (*evalml.pipelines.time_series_pipeline_base.TimeSeriesPipelineBase property*), 1622
custom_name (*evalml.pipelines.time_series_regression_pipeline.TimeSeriesRegressionPipeline property*), 1629
custom_name (*evalml.pipelines.TimeSeriesBinaryClassificationPipeline property*), 1797
custom_name (*evalml.pipelines.TimeSeriesClassificationPipeline property*), 1804
custom_name (*evalml.pipelines.TimeSeriesMulticlassClassificationPipeline property*), 1817
custom_name (*evalml.pipelines.TimeSeriesRegressionPipeline property*), 1825


## D


DaskComputation (*class in evalml.automl.engine.dask_engine*), 242
DaskEngine (*class in evalml.automl.engine*), 252
DaskEngine (*class in evalml.automl.engine.dask_engine*), 242
DataCheck (*class in evalml.data_checks*), 345
DataCheck (*class in evalml.data_checks.data_check*), 289
DataCheckAction (*class in evalml.data_checks*), 345
DataCheckAction (*class in evalml.data_checks.data_check_action*), 290
DataCheckActionCode (*class in evalml.data_checks*), 346
DataCheckActionCode (*class in evalml.data_checks.data_check_action_code*), 291
DataCheckActionOption (*class in evalml.data_checks*), 346
DataCheckActionOption (*class in evalml.data_checks.data_check_action_option*), 292
DataCheckError (*class in evalml.data_checks*), 348
DataCheckError (*class in evalml.data_checks.data_check_message*), 295
DataCheckInitError, 392, 395
DataCheckMessage (*class in evalml.data_checks*), 348

DataCheckMessage (*class in evalml.data_checks.data_check_message*), 295
DataCheckMessageCode (*class in evalml.data_checks*), 348
DataCheckMessageCode (*class in evalml.data_checks.data_check_message_code*), 296
DataCheckMessageType (*class in evalml.data_checks*), 350
DataCheckMessageType (*class in evalml.data_checks.data_check_message_type*), 298
DataChecks (*class in evalml.data_checks*), 351
DataChecks (*class in evalml.data_checks.data_checks*), 299
DataCheckWarning (*class in evalml.data_checks*), 351
DataCheckWarning (*class in evalml.data_checks.data_check_message*), 295
dates_needed_for_prediction() (*evalml.pipelines.time_series_classification_pipelines.TimeSeriesBinaryClassificationPipeline method*), 1599
dates_needed_for_prediction() (*evalml.pipelines.time_series_classification_pipelines.TimeSeriesClassificationPipeline method*), 1606
dates_needed_for_prediction() (*evalml.pipelines.time_series_classification_pipelines.TimeSeriesMulticlassClassificationPipeline method*), 1614
dates_needed_for_prediction() (*evalml.pipelines.time_series_pipeline_base.TimeSeriesPipelineBase method*), 1622
dates_needed_for_prediction() (*evalml.pipelines.time_series_regression_pipeline.TimeSeriesRegressionPipeline method*), 1629
dates_needed_for_prediction() (*evalml.pipelines.TimeSeriesBinaryClassificationPipeline method*), 1797
dates_needed_for_prediction() (*evalml.pipelines.TimeSeriesClassificationPipeline method*), 1804
dates_needed_for_prediction() (*evalml.pipelines.TimeSeriesMulticlassClassificationPipeline method*), 1817
dates_needed_for_prediction() (*evalml.pipelines.TimeSeriesRegressionPipeline method*), 1825
dates_needed_for_prediction_range() (*evalml.pipelines.time_series_classification_pipelines.TimeSeriesBinaryClassificationPipeline method*), 1599
dates_needed_for_prediction_range() (*evalml.pipelines.time_series_classification_pipelines.TimeSeriesClassificationPipeline method*), 1606
dates_needed_for_prediction_range() (*evalml.pipelines.time_series_classification_pipelines.TimeSeriesMulticlassClassificationPipeline method*), 1614
dates_needed_for_prediction_range() (*evalml.pipelines.time_series_pipeline_base.TimeSeriesPipelineBase method*), 1622
dates_needed_for_prediction_range() (*evalml.pipelines.time_series_regression_pipeline.TimeSeriesRegressionPipeline method*), 1629
dates_needed_for_prediction_range() (*evalml.pipelines.TimeSeriesBinaryClassificationPipeline method*), 1797
dates_needed_for_prediction_range() (*evalml.pipelines.TimeSeriesClassificationPipeline method*), 1804
dates_needed_for_prediction_range() (*evalml.pipelines.TimeSeriesMulticlassClassificationPipeline method*), 1817
dates_needed_for_prediction_range() (*evalml.pipelines.TimeSeriesRegressionPipeline method*), 1825
DateTimeFeaturizer (*class in evalml.pipelines.components*), 1256
DateTimeFeaturizer (*class in evalml.pipelines.components.transformers*), 1215
DateTimeFeaturizer (*class in evalml.pipelines.components.transformers.preprocessing*), 1128
DateTimeFeaturizer (*class in evalml.pipelines.components.transformers.preprocessing.datetime*), 1068
DateTimeFormatDataCheck (*class in evalml.data_checks*), 351
DateTimeFormatDataCheck (*class in evalml.data_checks.datetime_format_data_check*), 300
DCAOParameterAllowedValuesType (*class in evalml.data_checks*), 358
DCAOParameterAllowedValuesType (*class in evalml.data_checks.data_check_action_option*), 293
DCAOParameterType (*class in evalml.data_checks*), 359
DCAOParameterType (*class in evalml.data_checks.data_check_action_option*), 293
debug() (*evalml.automl.engine.engine_base.JobLogger method*), 246
decision_function() (*evalml.objectives.AUC method*), 535
decision_function() (*evalml.objectives.AUCSeriesClassificationPipeline*), 539
decision_function() (*evalml.objectives.BalancedAccuracyBinary*

*method*), 547
decision_function() (*evalml.objectives.binary_classification_objective.BinaryClassificationObjective method*), 441
decision_function() (*evalml.objectives.BinaryClassificationObjective method*), 550
decision_function() (*evalml.objectives.cost_benefit_matrix.CostBenefitMatrix method*), 445
decision_function() (*evalml.objectives.CostBenefitMatrix method*), 553
decision_function() (*evalml.objectives.F1 method*), 557
decision_function() (*evalml.objectives.fraud_cost.FraudCost method*), 448
decision_function() (*evalml.objectives.FraudCost method*), 565
decision_function() (*evalml.objectives.Gini method*), 569
decision_function() (*evalml.objectives.lead_scoring.LeadScoring method*), 450
decision_function() (*evalml.objectives.LeadScoring method*), 571
decision_function() (*evalml.objectives.LogLossBinary method*), 573
decision_function() (*evalml.objectives.MCCBinary method*), 582
decision_function() (*evalml.objectives.Precision method*), 596
decision_function() (*evalml.objectives.Recall method*), 605
decision_function() (*evalml.objectives.sensitivity_low_alert.SensitivityLowAlert method*), 461
decision_function() (*evalml.objectives.SensitivityLowAlert method*), 617
decision_function() (*evalml.objectives.standard_metrics.AccuracyBinary method*), 464
decision_function() (*evalml.objectives.standard_metrics.AUC method*), 468
decision_function() (*evalml.objectives.standard_metrics.BalancedAccuracyBinary method*), 476
decision_function() (*evalml.objectives.standard_metrics.F1 method*), 481
decision_function() (*evalml.objectives.standard_metrics.Gini method*), 489
decision_function() (*evalml.objectives.standard_metrics.LogLossBinary method*), 491
decision_function() (*evalml.objectives.standard_metrics.MCCBinary method*), 500
decision_function() (*evalml.objectives.standard_metrics.Precision method*), 509
decision_function() (*evalml.objectives.standard_metrics.Recall method*), 518
decision_tree_data_from_estimator() (*in module evalml.model_understanding.visualizations*), 421
decision_tree_data_from_pipeline() (*in module evalml.model_understanding.visualizations*), 422
DecisionTreeClassifier (*class in evalml.pipelines*), 1670
DecisionTreeClassifier (*class in evalml.pipelines.components*), 1359
DecisionTreeClassifier (*class in evalml.pipelines.components.estimators*), 875
DecisionTreeClassifier (*class in evalml.pipelines.components.estimators.classifiers*), 702
DecisionTreeClassifier (*class in evalml.pipelines.components.estimators.classifiers.decision_tree_*, 650
DecisionTreeRegressor (*class in evalml.pipelines*), 1674
DecisionTreeRegressor (*class in evalml.pipelines.components*), 1362
DecisionTreeRegressor (*class in evalml.pipelines.components.estimators*), 879
DecisionTreeRegressor (*class in evalml.pipelines.components.estimators.regressors*), 812
DecisionTreeRegressor (*class in evalml.pipelines.components.estimators.regressors.decision_tree_*, 754
Decomposer (*class in evalml.pipelines.components.transformers.preprocess*, 1131
Decomposer (*class in evalml.pipelines.components.transformers.preprocess*, 1071
DECOMPOSER_PERIOD_CAP (*in module evalml.pipelines.utils*), 1636
default_max_batches

(*evalml.automl.automl_algorithm.automl_algorithm.AutoMLAlgorithm property*), 224
default_max_batches (*evalml.automl.automl_algorithm.AutoMLAlgorithm property*), 233
default_max_batches (*evalml.automl.automl_algorithm.default_algorithm.DefaultAlgorithm property*), 227
default_max_batches (*evalml.automl.automl_algorithm.DefaultAlgorithm property*), 235
default_max_batches (*evalml.automl.automl_algorithm.iterative_algorithm.IterativeAlgorithm property*), 231
default_max_batches (*evalml.automl.automl_algorithm.IterativeAlgorithm property*), 238
DEFAULT_METRICS (*in module evalml.model_understanding.prediction_explanations.explainers*), 403
default_parameters (*evalml.pipelines.component_graph.ComponentGraph property*), 1570
default_parameters (*evalml.pipelines.ComponentGraph property*), 1667
default_parameters() (*evalml.pipelines.ARIMARegressor method*), 1643
default_parameters() (*evalml.pipelines.CatBoostClassifier method*), 1653
default_parameters() (*evalml.pipelines.CatBoostRegressor method*), 1657
default_parameters() (*evalml.pipelines.components.ARIMARegressor method*), 1338
default_parameters() (*evalml.pipelines.components.BaselineClassifier method*), 1341
default_parameters() (*evalml.pipelines.components.BaselineRegressor method*), 1344
default_parameters() (*evalml.pipelines.components.CatBoostClassifier method*), 1348
default_parameters() (*evalml.pipelines.components.CatBoostRegressor method*), 1351
default_parameters() (*evalml.pipelines.components.component_base.ComponentBase method*), 1324
default_parameters() (*evalml.pipelines.components.ComponentBase method*), 1354
default_parameters() (*evalml.pipelines.components.DateTimeFeaturizer method*), 1357
default_parameters() (*evalml.pipelines.components.DecisionTreeClassifier method*), 1360
default_parameters() (*evalml.pipelines.components.DecisionTreeRegressor method*), 1364
default_parameters() (*evalml.pipelines.components.DFSTransformer method*), 1368
default_parameters() (*evalml.pipelines.components.DropColumns method*), 1370
default_parameters() (*evalml.pipelines.components.DropNaNRowsTransformer method*), 1372
default_parameters() (*evalml.pipelines.components.DropNullColumns method*), 1375
default_parameters() (*evalml.pipelines.components.DropRowsTransformer method*), 1377
default_parameters() (*evalml.pipelines.components.ElasticNetClassifier method*), 1380
default_parameters() (*evalml.pipelines.components.ElasticNetRegressor method*), 1383
default_parameters() (*evalml.pipelines.components.EmailFeaturizer method*), 1386
default_parameters() (*evalml.pipelines.components.ensemble.stacked_ensemble_base.S method*), 620
default_parameters() (*evalml.pipelines.components.ensemble.stacked_ensemble_classif method*), 624
default_parameters() (*evalml.pipelines.components.ensemble.stacked_ensemble_regres method*), 629
default_parameters() (*evalml.pipelines.components.ensemble.StackedEnsembleBase method*), 632
default_parameters() (*evalml.pipelines.components.ensemble.StackedEnsembleClassifie method*), 636
default_parameters() (*evalml.pipelines.components.ensemble.StackedEnsembleRegress method*), 640
default_parameters() (*evalml.pipelines.components.Estimator method*), 1389

default_parameters() (evalml.pipelines.components.estimators.ARIMARegressor method), 860
default_parameters() (evalml.pipelines.components.estimators.BaselineClassifier method), 863
default_parameters() (evalml.pipelines.components.estimators.BaselineRegressor method), 866
default_parameters() (evalml.pipelines.components.estimators.CatBoostClassifier method), 870
default_parameters() (evalml.pipelines.components.estimators.CatBoostRegressor method), 873
default_parameters() (evalml.pipelines.components.estimators.classifiers.baseline_classifier.BaselineClassifier method), 644
default_parameters() (evalml.pipelines.components.estimators.classifiers.BaselineClassifier method), 697
default_parameters() (evalml.pipelines.components.estimators.classifiers.catboost_classifier.CatBoostClassifier method), 648
default_parameters() (evalml.pipelines.components.estimators.classifiers.CatBoostClassifier method), 700
default_parameters() (evalml.pipelines.components.estimators.classifiers.decision_tree_classifier.DecisionTreeClassifier method), 652
default_parameters() (evalml.pipelines.components.estimators.classifiers.DecisionTreeClassifier method), 704
default_parameters() (evalml.pipelines.components.estimators.classifiers.elastic_net_classifier.ElasticNetClassifier method), 656
default_parameters() (evalml.pipelines.components.estimators.classifiers.ElasticNetClassifier method), 708
default_parameters() (evalml.pipelines.components.estimators.classifiers.et_classifier_sklearn.ExtraTreesClassifier method), 660
default_parameters() (evalml.pipelines.components.estimators.classifiers.ExtraTreesClassifier method), 711
default_parameters() (evalml.pipelines.components.estimators.classifiers.kneighbors_classifier.KNeighborsClassifier method), 664
default_parameters() (evalml.pipelines.components.estimators.classifiers.KNeighborsClassifier method), 715
default_parameters() (evalml.pipelines.components.estimators.classifiers.lightgbm_classifier.LightGBMClassifier method), 669
default_parameters() (evalml.pipelines.components.estimators.classifiers.LightGBMCla method), 719
default_parameters() (evalml.pipelines.components.estimators.classifiers.logistic_regre method), 672
default_parameters() (evalml.pipelines.components.estimators.classifiers.LogisticRegre method), 722
default_parameters() (evalml.pipelines.components.estimators.classifiers.RandomFores method), 725
default_parameters() (evalml.pipelines.components.estimators.classifiers.rf_classifier.R method), 676
default_parameters() (evalml.pipelines.components.estimators.classifiers.svm_classifier.SVMClassifier method), 680
default_parameters() (evalml.pipelines.components.estimators.classifiers.SVMClassifie method), 729
default_parameters() (evalml.pipelines.components.estimators.classifiers.vowpal_wabb method), 683
default_parameters() (evalml.pipelines.components.estimators.classifiers.vowpal_wabb method), 686
default_parameters() (evalml.pipelines.components.estimators.classifiers.vowpal_wabb method), 690
default_parameters() (evalml.pipelines.components.estimators.classifiers.VowpalWabbi method), 732
default_parameters() (evalml.pipelines.components.estimators.classifiers.VowpalWabbi method), 735
default_parameters() (evalml.pipelines.components.estimators.classifiers.xgboost_class method), 694
default_parameters() (evalml.pipelines.components.estimators.classifiers.XGBoostClas method), 738
default_parameters() (evalml.pipelines.components.estimators.DecisionTreeClassifier method), 877
default_parameters() (evalml.pipelines.components.estimators.DecisionTreeRegressor method), 881
default_parameters() (evalml.pipelines.components.estimators.ElasticNetClassifier method), 884
default_parameters() (evalml.pipelines.components.estimators.ElasticNetRegressor method), 887

default_parameters() (evalml.pipelines.components.estimators.Estimator method), 891
default_parameters() (evalml.pipelines.components.estimators.estimator.Estimator method), 855
default_parameters() (evalml.pipelines.components.estimators.ExponentialSmoothingRegressor method), 894
default_parameters() (evalml.pipelines.components.estimators.ExtraTreesClassifier method), 898
default_parameters() (evalml.pipelines.components.estimators.ExtraTreesRegressor method), 902
default_parameters() (evalml.pipelines.components.estimators.KNeighborsClassifier method), 905
default_parameters() (evalml.pipelines.components.estimators.LightGBMClassifier method), 909
default_parameters() (evalml.pipelines.components.estimators.LightGBMRegressor method), 913
default_parameters() (evalml.pipelines.components.estimators.LinearRegressor method), 916
default_parameters() (evalml.pipelines.components.estimators.LogisticRegressionClassifier method), 919
default_parameters() (evalml.pipelines.components.estimators.ProphetRegressor method), 923
default_parameters() (evalml.pipelines.components.estimators.RandomForestClassifier method), 926
default_parameters() (evalml.pipelines.components.estimators.RandomForestRegressor method), 929
default_parameters() (evalml.pipelines.components.estimators.regressors.arima_regressor.ARIMARegressor method), 744
default_parameters() (evalml.pipelines.components.estimators.regressors.ARIMARegressor method), 803
default_parameters() (evalml.pipelines.components.estimators.regressors.baseline_regressor.BaselineRegressor method), 747
default_parameters() (evalml.pipelines.components.estimators.regressors.BaselineRegressor method), 806
default_parameters() (evalml.pipelines.components.estimators.regressors.catboost_regressor.CatBoostRegressor method), 751
default_parameters() (evalml.pipelines.components.estimators.regressors.CatBoostRegressor method), 810
default_parameters() (evalml.pipelines.components.estimators.regressors.decision_tree_regressor.DecisionTreeRegressor method), 755
default_parameters() (evalml.pipelines.components.estimators.regressors.DecisionTreeRegressor method), 814
default_parameters() (evalml.pipelines.components.estimators.regressors.elasticnet_regressor.ElasticNetRegressor method), 759
default_parameters() (evalml.pipelines.components.estimators.regressors.ElasticNetRegressor method), 817
default_parameters() (evalml.pipelines.components.estimators.regressors.et_regressor.ExtraTreesRegressor method), 764
default_parameters() (evalml.pipelines.components.estimators.regressors.exponential_smoothing_regressor.ExponentialSmoothingRegressor method), 768
default_parameters() (evalml.pipelines.components.estimators.regressors.ExponentialSmoothingRegressor method), 821
default_parameters() (evalml.pipelines.components.estimators.regressors.ExtraTreesRegressor method), 824
default_parameters() (evalml.pipelines.components.estimators.regressors.lightgbm_regressor.LightGBMRegressor method), 771
default_parameters() (evalml.pipelines.components.estimators.regressors.LightGBMRegressor method), 828
default_parameters() (evalml.pipelines.components.estimators.regressors.linear_regressor.LinearRegressor method), 775
default_parameters() (evalml.pipelines.components.estimators.regressors.LinearRegressor method), 831
default_parameters() (evalml.pipelines.components.estimators.regressors.prophet_regressor.ProphetRegressor method), 780
default_parameters() (evalml.pipelines.components.estimators.regressors.ProphetRegressor method), 835
default_parameters() (evalml.pipelines.components.estimators.regressors.RandomForestRegressor method), 838
default_parameters() (evalml.pipelines.components.estimators.regressors.rf_regressor.RandomForestRegressor method), 784
default_parameters() (evalml.pipelines.components.estimators.regressors.svm_regressor.SVMRegressor method), 787

default_parameters() (evalml.pipelines.components.estimators.regressors.SVMRegressor method), 841
default_parameters() (evalml.pipelines.components.estimators.regressors.time_series_baseline_estimator.TimeSeriesBaselineEstimator method), 791
default_parameters() (evalml.pipelines.components.estimators.regressors.TimeSeriesBaselineEstimator method), 844
default_parameters() (evalml.pipelines.components.estimators.regressors.vowpal_wabbit_regressor.VowpalWabbitBaseRegressor method), 794
default_parameters() (evalml.pipelines.components.estimators.regressors.VowpalWabbitRegressor method), 848
default_parameters() (evalml.pipelines.components.estimators.regressors.xgboost_regressor.XGBoostRegressor method), 798
default_parameters() (evalml.pipelines.components.estimators.regressors.XGBoostRegressor method), 851
default_parameters() (evalml.pipelines.components.estimators.SVMClassifier method), 932
default_parameters() (evalml.pipelines.components.estimators.SVMRegressor method), 935
default_parameters() (evalml.pipelines.components.estimators.TimeSeriesBaselineEstimator method), 939
default_parameters() (evalml.pipelines.components.estimators.VowpalWabbitBinaryClassifier method), 942
default_parameters() (evalml.pipelines.components.estimators.VowpalWabbitMulticlassClassifier method), 945
default_parameters() (evalml.pipelines.components.estimators.VowpalWabbitRegressor method), 948
default_parameters() (evalml.pipelines.components.estimators.XGBoostClassifier method), 951
default_parameters() (evalml.pipelines.components.estimators.XGBoostRegressor method), 955
default_parameters() (evalml.pipelines.components.ExponentialSmoothingRegressor method), 1392
default_parameters() (evalml.pipelines.components.ExtraTreesClassifier method), 1396
default_parameters() (evalml.pipelines.components.ExtraTreesRegressor method), 1400
default_parameters() (evalml.pipelines.components.FeatureSelector method), 1402
default_parameters() (evalml.pipelines.components.TimeSeriesBaselineEstimator method), 1405
default_parameters() (evalml.pipelines.components.KNeighborsClassifier method), 1408
default_parameters() (evalml.pipelines.components.VowpalWabbitBaseRegressor method), 1411
default_parameters() (evalml.pipelines.components.LightGBMClassifier method), 1414
default_parameters() (evalml.pipelines.components.LightGBMRegressor method), 1418
default_parameters() (evalml.pipelines.components.LinearDiscriminantAnalysis method), 1421
default_parameters() (evalml.pipelines.components.LinearRegressor method), 1423
default_parameters() (evalml.pipelines.components.LogisticRegressionClassifier method), 1427
default_parameters() (evalml.pipelines.components.LogTransformer method), 1430
default_parameters() (evalml.pipelines.components.LSA method), 1432
default_parameters() (evalml.pipelines.components.NaturalLanguageFeaturizer method), 1435
default_parameters() (evalml.pipelines.components.OneHotEncoder method), 1438
default_parameters() (evalml.pipelines.components.OrdinalEncoder method), 1441
default_parameters() (evalml.pipelines.components.Oversampler method), 1444
default_parameters() (evalml.pipelines.components.PCA method), 1447
default_parameters() (evalml.pipelines.components.PerColumnImputer method), 1449
default_parameters() (evalml.pipelines.components.PolynomialDecomposer method), 1452

default_parameters() (*evalml.pipelines.components.ProphetRegressor method*), 1457
default_parameters() (*evalml.pipelines.components.RandomForestClassifier method*), 1460
default_parameters() (*evalml.pipelines.components.RandomForestRegressor method*), 1463
default_parameters() (*evalml.pipelines.components.ReplaceNullableTypes method*), 1466
default_parameters() (*evalml.pipelines.components.RFClassifierRFESelector method*), 1469
default_parameters() (*evalml.pipelines.components.RFClassifierSelectFromModel method*), 1472
default_parameters() (*evalml.pipelines.components.RFRegressorRFESelector method*), 1475
default_parameters() (*evalml.pipelines.components.RFRegressorSelectFromModel method*), 1478
default_parameters() (*evalml.pipelines.components.SelectByType method*), 1481
default_parameters() (*evalml.pipelines.components.SelectColumns method*), 1483
default_parameters() (*evalml.pipelines.components.SimpleImputer method*), 1485
default_parameters() (*evalml.pipelines.components.StackedEnsembleBase method*), 1488
default_parameters() (*evalml.pipelines.components.StackedEnsembleClassifier method*), 1492
default_parameters() (*evalml.pipelines.components.StackedEnsembleRegressor method*), 1496
default_parameters() (*evalml.pipelines.components.StandardScaler method*), 1498
default_parameters() (*evalml.pipelines.components.STLDecomposer method*), 1501
default_parameters() (*evalml.pipelines.components.SVMClassifier method*), 1506
default_parameters() (*evalml.pipelines.components.SVMRegressor method*), 1509
default_parameters() (*evalml.pipelines.components.TargetEncoder method*), 1512
default_parameters() (*evalml.pipelines.components.TargetImputer method*), 1515
default_parameters() (*evalml.pipelines.components.TimeSeriesBaselineEstimator method*), 1518
default_parameters() (*evalml.pipelines.components.TimeSeriesFeaturizer method*), 1522
default_parameters() (*evalml.pipelines.components.TimeSeriesImputer method*), 1524
default_parameters() (*evalml.pipelines.components.TimeSeriesRegularizer method*), 1527
default_parameters() (*evalml.pipelines.components.Transformer method*), 1530
default_parameters() (*evalml.pipelines.components.transformers.column_selectors.Col method*), 1202
default_parameters() (*evalml.pipelines.components.transformers.column_selectors.Dro method*), 1204
default_parameters() (*evalml.pipelines.components.transformers.column_selectors.Sele method*), 1207
default_parameters() (*evalml.pipelines.components.transformers.column_selectors.Sele method*), 1209
default_parameters() (*evalml.pipelines.components.transformers.DateTimeFeaturizer method*), 1216
default_parameters() (*evalml.pipelines.components.transformers.DFSTransformer method*), 1219
default_parameters() (*evalml.pipelines.components.transformers.dimensionality_reduc method*), 958
default_parameters() (*evalml.pipelines.components.transformers.dimensionality_reduc method*), 964
default_parameters() (*evalml.pipelines.components.transformers.dimensionality_reduc method*), 966
default_parameters() (*evalml.pipelines.components.transformers.dimensionality_reduc method*), 961
default_parameters() (*evalml.pipelines.components.transformers.DropColumns method*), 1222

default_parameters() (evalml.pipelines.components.transformers.DropNaNRowsTransformer method), 1224
default_parameters() (evalml.pipelines.components.transformers.DropNullColumns method), 1227
default_parameters() (evalml.pipelines.components.transformers.DropRowsTransformer method), 1229
default_parameters() (evalml.pipelines.components.transformers.EmailFeaturizer method), 1231
default_parameters() (evalml.pipelines.components.transformers.encoders.label_encoder.LabelEncoder method), 969
default_parameters() (evalml.pipelines.components.transformers.encoders.LabelEncoder method), 985
default_parameters() (evalml.pipelines.components.transformers.encoders.onehot_encoder.OneHotEncoder method), 973
default_parameters() (evalml.pipelines.components.transformers.encoders.OneHotEncoder method), 988
default_parameters() (evalml.pipelines.components.transformers.encoders.ordinal_encoder.OrdinalEncoder method), 978
default_parameters() (evalml.pipelines.components.transformers.encoders.OrdinalEncoder method), 992
default_parameters() (evalml.pipelines.components.transformers.encoders.target_encoder.TargetEncoder method), 982
default_parameters() (evalml.pipelines.components.transformers.encoders.TargetEncoder method), 994
default_parameters() (evalml.pipelines.components.transformers.feature_selection.feature_selection.FeatureSelection method), 998
default_parameters() (evalml.pipelines.components.transformers.feature_selection.FeatureSelector method), 1018
default_parameters() (evalml.pipelines.components.transformers.feature_selection.recursive_elimination.RecursiveFeatureElimination method), 1001
default_parameters() (evalml.pipelines.components.transformers.feature_selection.recursive_elimination.RFCLassifierRFE method), 1004
default_parameters() (evalml.pipelines.components.transformers.feature_selection.recursive_elimination.RFRegressorRFE method), 1007
default_parameters() (evalml.pipelines.components.transformers.feature_selection.select_from_model.RFClassifierSelectFromModel method), 1011
default_parameters() (evalml.pipelines.components.transformers.feature_selection.rf_r method), 1014
default_parameters() (evalml.pipelines.components.transformers.feature_selection.RFC method), 1021
default_parameters() (evalml.pipelines.components.transformers.feature_selection.RFC method), 1024
default_parameters() (evalml.pipelines.components.transformers.feature_selection.RFR method), 1027
default_parameters() (evalml.pipelines.components.transformers.feature_selection.RFR method), 1030
default_parameters() (evalml.pipelines.components.transformers.FeatureSelector method), 1234
default_parameters() (evalml.pipelines.components.transformers.Imputer method), 1237
default_parameters() (evalml.pipelines.components.transformers.imputers.Imputer method), 1053
default_parameters() (evalml.pipelines.components.transformers.imputers.imputer.Impu method), 1034
default_parameters() (evalml.pipelines.components.transformers.imputers.knn_imputer method), 1037
default_parameters() (evalml.pipelines.components.transformers.imputers.KNNImputer method), 1056
default_parameters() (evalml.pipelines.components.transformers.imputers.per_column_ method), 1040
default_parameters() (evalml.pipelines.components.transformers.imputers.PerColumnI method), 1058
default_parameters() (evalml.pipelines.components.transformers.imputers.simple_impu method), 1043
default_parameters() (evalml.pipelines.components.transformers.imputers.SimpleImput method), 1060
default_parameters() (evalml.pipelines.components.transformers.RFClassifierRFE method), 1046
default_parameters() (evalml.pipelines.components.transformers.RFRegressorRFE method), 1063
default_parameters() (evalml.pipelines.components.transformers.SelectFromModel method), 1050

default_parameters() (evalml.pipelines.components.transformers.imputers.TimeSeriesImputer method), 1066
default_parameters() (evalml.pipelines.components.transformers.LabelEncoder method), 1239
default_parameters() (evalml.pipelines.components.transformers.LinearDiscriminantAnalysis method), 1242
default_parameters() (evalml.pipelines.components.transformers.LogTransformer method), 1244
default_parameters() (evalml.pipelines.components.transformers.LSA method), 1247
default_parameters() (evalml.pipelines.components.transformers.NaturalLanguageFeaturizer method), 1249
default_parameters() (evalml.pipelines.components.transformers.OneHotEncoder method), 1252
default_parameters() (evalml.pipelines.components.transformers.OrdinalEncoder method), 1256
default_parameters() (evalml.pipelines.components.transformers.Oversampler method), 1259
default_parameters() (evalml.pipelines.components.transformers.PCA method), 1261
default_parameters() (evalml.pipelines.components.transformers.PerColumnImputer method), 1264
default_parameters() (evalml.pipelines.components.transformers.PolynomialDecomposer method), 1267
default_parameters() (evalml.pipelines.components.transformers.preprocessing.datetime.DatetimeFeaturizer method), 1069
default_parameters() (evalml.pipelines.components.transformers.preprocessing.DateTimeFeaturizer method), 1129
default_parameters() (evalml.pipelines.components.transformers.preprocessing.DelayedFeatureTransformer method), 1132
default_parameters() (evalml.pipelines.components.transformers.preprocessing.drop_columns.DropColumns method), 1073
default_parameters() (evalml.pipelines.components.transformers.preprocessing.DROPCOLUMNS method), 1136
default_parameters() (evalml.pipelines.components.transformers.preprocessing.drop_nan_rows_transformer.DropNullRowsTransformer method), 1077
default_parameters() (evalml.pipelines.components.transformers.preprocessing.drop_n method), 1080
default_parameters() (evalml.pipelines.components.transformers.preprocessing.drop_r method), 1083
default_parameters() (evalml.pipelines.components.transformers.preprocessing.DropNa method), 1139
default_parameters() (evalml.pipelines.components.transformers.preprocessing.DropNu method), 1141
default_parameters() (evalml.pipelines.components.transformers.preprocessing.DropRo method), 1144
default_parameters() (evalml.pipelines.components.transformers.preprocessing.EmailF method), 1146
default_parameters() (evalml.pipelines.components.transformers.preprocessing.feature method), 1086
default_parameters() (evalml.pipelines.components.transformers.preprocessing.log_tra method), 1089
default_parameters() (evalml.pipelines.components.transformers.preprocessing.LogTra method), 1149
default_parameters() (evalml.pipelines.components.transformers.preprocessing.LSA method), 1151
default_parameters() (evalml.pipelines.components.transformers.preprocessing.lsa.LSA method), 1092
default_parameters() (evalml.pipelines.components.transformers.preprocessing.natural method), 1095
default_parameters() (evalml.pipelines.components.transformers.preprocessing.Natural method), 1153
default_parameters() (evalml.pipelines.components.transformers.preprocessing.polyno method), 1099
default_parameters() (evalml.pipelines.components.transformers.preprocessing.Polyno method), 1157
default_parameters() (evalml.pipelines.components.transformers.preprocessing.replace method), 1104
default_parameters() (evalml.pipelines.components.transformers.preprocessing.Replace method), 1161
default_parameters() (evalml.pipelines.components.transformers.preprocessing.stl_dec method), 1108

default_parameters() (evalml.pipelines.components.transformers.preprocessing.STLDecomposer method), 1164
default_parameters() (evalml.pipelines.components.transformers.preprocessing.text_transformer.TextTransformer method), 1113
default_parameters() (evalml.pipelines.components.transformers.preprocessing.TextVectorizer method), 1169
default_parameters() (evalml.pipelines.components.transformers.preprocessing.time_series_featurizer.TimeSeriesFeaturizer method), 1117
default_parameters() (evalml.pipelines.components.transformers.preprocessing.time_series_regularizer.TimeSeriesRegularizer method), 1120
default_parameters() (evalml.pipelines.components.transformers.preprocessing.TimeSeriesSelectorFeaturizer method), 1172
default_parameters() (evalml.pipelines.components.transformers.preprocessing.TimeSeriesRegularizer method), 1175
default_parameters() (evalml.pipelines.components.transformers.preprocessing.transformer_primitive_components.EmailFeaturizer method), 1123
default_parameters() (evalml.pipelines.components.transformers.preprocessing.transformer_primitive_components.URLFeaturizer method), 1126
default_parameters() (evalml.pipelines.components.transformers.preprocessing.URLFeaturizer method), 1178
default_parameters() (evalml.pipelines.components.transformers.ReplaceNullableTypes method), 1272
default_parameters() (evalml.pipelines.components.transformers.RFClassifierRFSelectFromModel method), 1274
default_parameters() (evalml.pipelines.components.transformers.RFClassifierSelectFromModel method), 1277
default_parameters() (evalml.pipelines.components.transformers.RFRegressorRFSelectFromModel method), 1280
default_parameters() (evalml.pipelines.components.transformers.RFRegressorSelectFromModel method), 1283
default_parameters() (evalml.pipelines.components.transformers.samplers.base_sampler.BaseSampler method), 1181
default_parameters() (evalml.pipelines.components.transformers.samplers.OversamplerMixin method), 1190
default_parameters() (evalml.pipelines.components.transformers.samplers.oversampler_mixin.OversamplerMixin method), 1184
default_parameters() (evalml.pipelines.components.transformers.samplers.UndersamplerMixin method), 1193
default_parameters() (evalml.pipelines.components.transformers.samplers.undersampler_mixin.UndersamplerMixin method), 1187
default_parameters() (evalml.pipelines.components.transformers.scalers.standard_scaler.StandardScaler method), 1196
default_parameters() (evalml.pipelines.components.transformers.scalers.StandardScaler method), 1199
default_parameters() (evalml.pipelines.components.transformers.SelectByType method), 1286
default_parameters() (evalml.pipelines.components.transformers.SelectColumns method), 1288
default_parameters() (evalml.pipelines.components.transformers.SimpleImputer method), 1290
default_parameters() (evalml.pipelines.components.transformers.StandardScaler method), 1293
default_parameters() (evalml.pipelines.components.transformers.STLDecomposer method), 1296
default_parameters() (evalml.pipelines.components.transformers.TargetEncoder method), 1301
default_parameters() (evalml.pipelines.components.transformers.TargetImputer method), 1304
default_parameters() (evalml.pipelines.components.transformers.TimeSeriesFeaturizer method), 1307
default_parameters() (evalml.pipelines.components.transformers.TimeSeriesImputer method), 1310
default_parameters() (evalml.pipelines.components.transformers.TimeSeriesRegularizer method), 1313
default_parameters() (evalml.pipelines.components.transformers.Transformer method), 1316
default_parameters() (evalml.pipelines.components.transformers.transformer.Transformer method), 1212
default_parameters() (evalml.pipelines.components.transformers.Undersampler method), 1319
default_parameters() (evalml.pipelines.components.transformers.URLFeaturizer method), 1322

default_parameters() (*evalml.pipelines.components.Undersampler method*), 1533
default_parameters() (*evalml.pipelines.components.URLFeaturizer method*), 1535
default_parameters() (*evalml.pipelines.components.VowpalWabbitBinaryClassifier method*), 1538
default_parameters() (*evalml.pipelines.components.VowpalWabbitMulticlassClassifier method*), 1541
default_parameters() (*evalml.pipelines.components.VowpalWabbitRegressor method*), 1544
default_parameters() (*evalml.pipelines.components.XGBoostClassifier method*), 1547
default_parameters() (*evalml.pipelines.components.XGBoostRegressor method*), 1550
default_parameters() (*evalml.pipelines.DecisionTreeClassifier method*), 1672
default_parameters() (*evalml.pipelines.DecisionTreeRegressor method*), 1676
default_parameters() (*evalml.pipelines.DFSTransformer method*), 1679
default_parameters() (*evalml.pipelines.DropNaNRowsTransformer method*), 1682
default_parameters() (*evalml.pipelines.ElasticNetClassifier method*), 1685
default_parameters() (*evalml.pipelines.ElasticNetRegressor method*), 1688
default_parameters() (*evalml.pipelines.Estimator method*), 1691
default_parameters() (*evalml.pipelines.ExponentialSmoothingRegressor method*), 1694
default_parameters() (*evalml.pipelines.ExtraTreesClassifier method*), 1698
default_parameters() (*evalml.pipelines.ExtraTreesRegressor method*), 1702
default_parameters() (*evalml.pipelines.FeatureSelector method*), 1705
default_parameters() (*evalml.pipelines.Imputer method*), 1708
default_parameters() (*evalml.pipelines.KNeighborsClassifier method*), 1711
default_parameters() (*evalml.pipelines.LightGBMClassifier method*), 1714
default_parameters() (*evalml.pipelines.LightGBMRegressor method*), 1718
default_parameters() (*evalml.pipelines.LinearRegressor method*), 1721
default_parameters() (*evalml.pipelines.LogisticRegressionClassifier method*), 1724
default_parameters() (*evalml.pipelines.OneHotEncoder method*), 1734
default_parameters() (*evalml.pipelines.OrdinalEncoder method*), 1738
default_parameters() (*evalml.pipelines.PerColumnImputer method*), 1740
default_parameters() (*evalml.pipelines.ProphetRegressor method*), 1749
default_parameters() (*evalml.pipelines.RandomForestClassifier method*), 1752
default_parameters() (*evalml.pipelines.RandomForestRegressor method*), 1755
default_parameters() (*evalml.pipelines.RFClassifierSelectFromModel method*), 1764
default_parameters() (*evalml.pipelines.RFRegressorSelectFromModel method*), 1767
default_parameters() (*evalml.pipelines.SimpleImputer method*), 1770
default_parameters() (*evalml.pipelines.StackedEnsembleBase method*), 1773
default_parameters() (*evalml.pipelines.StackedEnsembleClassifier method*), 1777
default_parameters() (*evalml.pipelines.StackedEnsembleRegressor method*), 1781
default_parameters() (*evalml.pipelines.StandardScaler method*),

1783
default_parameters() (evalml.pipelines.SVMClassifier method), 1786
default_parameters() (evalml.pipelines.SVMRegressor method), 1789
default_parameters() (evalml.pipelines.TargetEncoder method), 1792
default_parameters() (evalml.pipelines.TimeSeriesFeaturizer method), 1811
default_parameters() (evalml.pipelines.TimeSeriesImputer method), 1813
default_parameters() (evalml.pipelines.TimeSeriesRegularizer method), 1832
default_parameters() (evalml.pipelines.Transformer method), 1835
default_parameters() (evalml.pipelines.VowpalWabbitBinaryClassifier method), 1838
default_parameters() (evalml.pipelines.VowpalWabbitMulticlassClassifier method), 1841
default_parameters() (evalml.pipelines.VowpalWabbitRegressor method), 1844
default_parameters() (evalml.pipelines.XGBoostClassifier method), 1847
default_parameters() (evalml.pipelines.XGBoostRegressor method), 1850
DefaultAlgorithm (class in evalml.automl.automl_algorithm), 233
DefaultAlgorithm (class in evalml.automl.automl_algorithm.default_algorithm), 225
DefaultDataChecks (class in evalml.data_checks), 360
DefaultDataChecks (class in evalml.data_checks.default_data_checks), 307
deprecate_arg() (in module evalml.utils), 1910
deprecate_arg() (in module evalml.utils.gen_utils), 1903
describe() (evalml.pipelines.ARIMARegressor method), 1643
describe() (evalml.pipelines.binary_classification_pipeline.BinaryClassificationPipeline method), 1556
describe() (evalml.pipelines.BinaryClassificationPipeline method), 1648
describe() (evalml.pipelines.CatBoostClassifier method), 1654
describe() (evalml.pipelines.CatBoostRegressor method), 1657
describe() (evalml.pipelines.classification_pipeline.ClassificationPipelin method), 1564
describe() (evalml.pipelines.ClassificationPipeline method), 1661
describe() (evalml.pipelines.component_graph.ComponentGraph method), 1570
describe() (evalml.pipelines.ComponentGraph method), 1667
describe() (evalml.pipelines.components.ARIMARegressor method), 1338
describe() (evalml.pipelines.components.BaselineClassifier method), 1342
describe() (evalml.pipelines.components.BaselineRegressor method), 1344
describe() (evalml.pipelines.components.CatBoostClassifier method), 1348
describe() (evalml.pipelines.components.CatBoostRegressor method), 1351
describe() (evalml.pipelines.components.component_base.ComponentBa method), 1325
describe() (evalml.pipelines.components.ComponentBase method), 1354
describe() (evalml.pipelines.components.DateTimeFeaturizer method), 1357
describe() (evalml.pipelines.components.DecisionTreeClassifier method), 1361
describe() (evalml.pipelines.components.DecisionTreeRegressor method), 1364
describe() (evalml.pipelines.components.DFSTransformer method), 1368
describe() (evalml.pipelines.components.DropColumns method), 1370
describe() (evalml.pipelines.components.DropNaNRowsTransformer method), 1372
describe() (evalml.pipelines.components.DropNullColumns method), 1375
describe() (evalml.pipelines.components.DropRowsTransformer method), 1377
describe() (evalml.pipelines.components.ElasticNetClassifier method), 1380
describe() (evalml.pipelines.components.ElasticNetRegressor method), 1383
describe() (evalml.pipelines.components.EmailFeaturizer method), 1386
describe() (evalml.pipelines.components.ensemble.stacked_ensemble_ba method), 620
describe() (evalml.pipelines.components.ensemble.stacked_ensemble_cla method), 625
describe() (evalml.pipelines.components.ensemble.stacked_ensemble_reg method), 629

describe() (*evalml.pipelines.components.ensemble.StackedEnsembleBase method*), 632
describe() (*evalml.pipelines.components.ensemble.StackedEnsembleClassifier method*), 636
describe() (*evalml.pipelines.components.ensemble.StackedEnsembleRegressor method*), 640
describe() (*evalml.pipelines.components.Estimator method*), 1389
describe() (*evalml.pipelines.components.estimators.ARIMARegressor method*), 860
describe() (*evalml.pipelines.components.estimators.BaselineClassifier method*), 864
describe() (*evalml.pipelines.components.estimators.BaselineRegressor method*), 867
describe() (*evalml.pipelines.components.estimators.CatBoostClassifier method*), 870
describe() (*evalml.pipelines.components.estimators.CatBoostRegressor method*), 873
describe() (*evalml.pipelines.components.estimators.classifiers.BaselineClassifier method*), 644
describe() (*evalml.pipelines.components.estimators.classifiers.BinaryClassifier method*), 697
describe() (*evalml.pipelines.components.estimators.classifiers.CatBoostClassifier method*), 648
describe() (*evalml.pipelines.components.estimators.classifiers.CatBoostClassifier method*), 701
describe() (*evalml.pipelines.components.estimators.classifiers.DecisionTreeClassifier method*), 652
describe() (*evalml.pipelines.components.estimators.classifiers.DecisionTreeClassifier method*), 704
describe() (*evalml.pipelines.components.estimators.classifiers.ElasticNetClassifier method*), 656
describe() (*evalml.pipelines.components.estimators.classifiers.ElasticNetClassifier method*), 708
describe() (*evalml.pipelines.components.estimators.classifiers.ExtraTreesClassifier method*), 660
describe() (*evalml.pipelines.components.estimators.classifiers.ExtraTreesClassifier method*), 711
describe() (*evalml.pipelines.components.estimators.classifiers.KNeighborsClassifier method*), 665
describe() (*evalml.pipelines.components.estimators.classifiers.KNeighborsClassifier method*), 715
describe() (*evalml.pipelines.components.estimators.classifiers.LightGBMClassifier method*), 669
describe() (*evalml.pipelines.components.estimators.classifiers.LightGBMClassifier method*), 719
describe() (*evalml.pipelines.components.estimators.classifiers.LogisticRegressionClassifier method*), 672
describe() (*evalml.pipelines.components.estimators.classifiers.RandomForestClassifier method*), 722
describe() (*evalml.pipelines.components.estimators.classifiers.RandomForestClassifier method*), 725
describe() (*evalml.pipelines.components.estimators.classifiers.RandomForestClassifier method*), 676
describe() (*evalml.pipelines.components.estimators.classifiers.svm_classifier.BaseSVMClassifier method*), 680
describe() (*evalml.pipelines.components.estimators.classifiers.SVMClassifier method*), 729
describe() (*evalml.pipelines.components.estimators.classifiers.vowpal_wabbit_base.VowpalWabbitBaseClassifier method*), 683
describe() (*evalml.pipelines.components.estimators.classifiers.vowpal_wabbit_base.VowpalWabbitBaseClassifier method*), 687
describe() (*evalml.pipelines.components.estimators.classifiers.vowpal_wabbit_base.VowpalWabbitBaseRegressor method*), 690
describe() (*evalml.pipelines.components.estimators.classifiers.VowpalWabbitBinaryClassifier method*), 732
describe() (*evalml.pipelines.components.estimators.classifiers.VowpalWabbitMulticlassClassifier method*), 735
describe() (*evalml.pipelines.components.estimators.classifiers.xgboost_classifier.XGBoostClassifier method*), 694
describe() (*evalml.pipelines.components.estimators.classifiers.XGBoostClassifier method*), 739
describe() (*evalml.pipelines.components.estimators.DecisionTreeClassifier method*), 877
describe() (*evalml.pipelines.components.estimators.DecisionTreeRegressor method*), 881
describe() (*evalml.pipelines.components.estimators.ElasticNetClassifier method*), 885
describe() (*evalml.pipelines.components.estimators.ElasticNetRegressor method*), 888
describe() (*evalml.pipelines.components.estimators.Estimator method*), 891
describe() (*evalml.pipelines.components.estimator.Estimator method*), 855
describe() (*evalml.pipelines.components.estimators.ExponentialSmoothingRegressor method*), 894
describe() (*evalml.pipelines.components.estimators.ExtraTreesClassifier method*), 898
describe() (*evalml.pipelines.components.estimators.ExtraTreesRegressor method*), 902
describe() (*evalml.pipelines.components.estimators.KNeighborsClassifier method*), 906
describe() (*evalml.pipelines.components.estimators.LightGBMClassifier method*), 909
describe() (*evalml.pipelines.components.estimators.LightGBMRegressor method*), 913
describe() (*evalml.pipelines.components.estimators.LinearRegressor method*), 916
describe() (*evalml.pipelines.components.estimators.LogisticRegressionClassifier method*), 920
describe() (*evalml.pipelines.components.estimators.LogisticRegressionCrossfitRegressor method*), 923
describe() (*evalml.pipelines.components.estimators.RandomForestClassifier method*), 926
describe() (*evalml.pipelines.components.estimators.RandomForestRegressor method*), 929
describe() (*evalml.pipelines.components.estimators.regressors.arima_regressor method*), 744

describe() (evalml.pipelines.components.estimators.regressors.ARIMARegressor method), 803
describe() (evalml.pipelines.components.estimators.regressors.BaselineRegressor method), 747
describe() (evalml.pipelines.components.estimators.regressors.CatBoostRegressor method), 806
describe() (evalml.pipelines.components.estimators.regressors.DecisionTreeRegressor method), 751
describe() (evalml.pipelines.components.estimators.regressors.ElasticNetRegressor method), 810
describe() (evalml.pipelines.components.estimators.regressors.ExtraTreesRegressor method), 755
describe() (evalml.pipelines.components.estimators.regressors.GradientBoostRegressor method), 814
describe() (evalml.pipelines.components.estimators.regressors.KNeighborsRegressor method), 759
describe() (evalml.pipelines.components.estimators.regressors.LinearRegressor method), 817
describe() (evalml.pipelines.components.estimators.regressors.LightGBMRegressor method), 764
describe() (evalml.pipelines.components.estimators.regressors.OrdinalRegressor method), 768
describe() (evalml.pipelines.components.estimators.regressors.RandomForestRegressor method), 821
describe() (evalml.pipelines.components.estimators.regressors.RollingMeanRegressor method), 824
describe() (evalml.pipelines.components.estimators.regressors.SVMRegressor method), 772
describe() (evalml.pipelines.components.estimators.regressors.XGBoostRegressor method), 828
describe() (evalml.pipelines.components.estimators.regressors.baseline_regressor.BaselineRegressor method), 775
describe() (evalml.pipelines.components.estimators.regressors.arima.ARIMARegressor method), 831
describe() (evalml.pipelines.components.estimators.regressors.catboost_regressor.CatBoostRegressor method), 780
describe() (evalml.pipelines.components.estimators.regressors.decision_tree_regressor.DecisionTreeRegressor method), 835
describe() (evalml.pipelines.components.estimators.regressors.elasticnet_regressor.ElasticNetRegressor method), 838
describe() (evalml.pipelines.components.estimators.regressors.extra_trees_regressor.ExtraTreesRegressor method), 784
describe() (evalml.pipelines.components.estimators.regressors.gradient_boost_regressor.GradientBoostRegressor method), 787
describe() (evalml.pipelines.components.estimators.regressors.knn_regressor.KNeighborsRegressor method), 841
describe() (evalml.pipelines.components.estimators.regressors.linear_regressor.LinearRegressor method), 791
describe() (evalml.pipelines.components.estimators.regressors.lightgbm_regressor.LightGBMRegressor method), 845
describe() (evalml.pipelines.components.estimators.regressors.ordinal_regressor.OrdinalRegressor method), 795
describe() (evalml.pipelines.components.estimators.regressors.rolling_mean_regressor.RollingMeanRegressor method), 848
describe() (evalml.pipelines.components.estimators.regressors.svm_regressor.SVMRegressor method), 799
describe() (evalml.pipelines.components.estimators.regressors.xgboost_regressor.XGBoostRegressor method), 851
describe() (evalml.pipelines.components.estimators.SVMClassifier method), 932
describe() (evalml.pipelines.components.estimators.SVMRegressor method), 936
describe() (evalml.pipelines.components.estimators.TimeSeriesBaselineEstimator method), 939
describe() (evalml.pipelines.components.estimators.VowpalWabbitBinaryClassifier method), 942
describe() (evalml.pipelines.components.estimators.VowpalWabbitMulticlassClassifier method), 945
describe() (evalml.pipelines.components.estimators.VowpalWabbitRegressor method), 948
describe() (evalml.pipelines.components.estimators.XGBoostClassifier method), 952
describe() (evalml.pipelines.components.estimators.XGBoostRegressor method), 955
describe() (evalml.pipelines.regression.ExponentialSmoothingRegressor method), 1392
describe() (evalml.pipelines.components.ExtraTreesClassifier method), 1396
describe() (evalml.pipelines.components.ExtraTreesRegressor method), 1400
describe() (evalml.pipelines.components.RandomFeatureSelector method), 1403
describe() (evalml.pipelines.components.Imputer method), 1405
describe() (evalml.pipelines.components.KNeighborsClassifier method), 1408
describe() (evalml.pipelines.components.LabelEncoder method), 1411
describe() (evalml.pipelines.components.LightGBMClassifier method), 1415
describe() (evalml.pipelines.components.LightGBMRegressor method), 1418
describe() (evalml.pipelines.components.LinearDiscriminantAnalysis method), 1421
describe() (evalml.pipelines.components.LinearRegressor method), 1424
describe() (evalml.pipelines.components.LogisticRegressionClassifier method), 1427
describe() (evalml.pipelines.components.LogTransformer method), 1430
describe() (evalml.pipelines.components.TimeSeriesBaselineEstimator method), 1433
describe() (evalml.pipelines.components.NaturalLanguageFeaturizer method), 1435
describe() (evalml.pipelines.components.OneHotEncoder method), 1438
describe() (evalml.pipelines.components.OrdinalEncoder method), 1442

describe() (*evalml.pipelines.components.Oversampler method*), 1444
describe() (*evalml.pipelines.components.PCA method*), 1447
describe() (*evalml.pipelines.components.PerColumnImputer method*), 1449
describe() (*evalml.pipelines.components.PolynomialDecomposer method*), 1452
describe() (*evalml.pipelines.components.ProphetRegressor method*), 1457
describe() (*evalml.pipelines.components.RandomForestClassifier method*), 1460
describe() (*evalml.pipelines.components.RandomForestRegressor method*), 1463
describe() (*evalml.pipelines.components.ReplaceNullableTypes method*), 1466
describe() (*evalml.pipelines.components.RFClassifierRFESelector method*), 1469
describe() (*evalml.pipelines.components.RFClassifierSelectFromModel method*), 1472
describe() (*evalml.pipelines.components.RFRegressorRFESelector method*), 1475
describe() (*evalml.pipelines.components.RFRegressorSelectFromModel method*), 1478
describe() (*evalml.pipelines.components.SelectByType method*), 1481
describe() (*evalml.pipelines.components.SelectColumns method*), 1483
describe() (*evalml.pipelines.components.SimpleImputer method*), 1485
describe() (*evalml.pipelines.components.StackedEnsembleBase method*), 1488
describe() (*evalml.pipelines.components.StackedEnsembleClassifier method*), 1492
describe() (*evalml.pipelines.components.StackedEnsembleRegressor method*), 1496
describe() (*evalml.pipelines.components.StandardScaler method*), 1498
describe() (*evalml.pipelines.components.STLDecomposer method*), 1501
describe() (*evalml.pipelines.components.SVMClassifier method*), 1506
describe() (*evalml.pipelines.components.SVMRegressor method*), 1510
describe() (*evalml.pipelines.components.TargetEncoder method*), 1513
describe() (*evalml.pipelines.components.TargetImputer method*), 1515
describe() (*evalml.pipelines.components.TimeSeriesBaselineEstimator method*), 1518
describe() (*evalml.pipelines.components.TimeSeriesFeaturizer method*), 1522
describe() (*evalml.pipelines.components.TimeSeriesImputer method*), 1524
describe() (*evalml.pipelines.components.TimeSeriesRegularizer method*), 1527
describe() (*evalml.pipelines.components.Transformer method*), 1530
describe() (*evalml.pipelines.components.transformers.column_selectors. method*), 1202
describe() (*evalml.pipelines.components.transformers.column_selectors. method*), 1205
describe() (*evalml.pipelines.components.transformers.column_selectors. method*), 1207
describe() (*evalml.pipelines.components.transformers.column_selectors. method*), 1209
describe() (*evalml.pipelines.components.transformers.DateTimeFeaturiz method*), 1216
describe() (*evalml.pipelines.components.transformers.DFSTransformer method*), 1219
describe() (*evalml.pipelines.components.transformers.dimensionality_re method*), 958
describe() (*evalml.pipelines.components.transformers.dimensionality_re method*), 964
describe() (*evalml.pipelines.components.transformers.dimensionality_re method*), 966
describe() (*evalml.pipelines.components.transformers.dimensionality_re method*), 961
describe() (*evalml.pipelines.components.transformers.DropColumns method*), 1222
describe() (*evalml.pipelines.components.transformers.DropNaNRowsTra method*), 1224
describe() (*evalml.pipelines.components.transformers.DropNullColumns method*), 1227
describe() (*evalml.pipelines.components.transformers.DropRowsTransfo method*), 1229
describe() (*evalml.pipelines.components.transformers.EmailFeaturizer method*), 1231
describe() (*evalml.pipelines.components.transformers.encoders.label_en method*), 969
describe() (*evalml.pipelines.components.transformers.encoders.LabelEn method*), 985
describe() (*evalml.pipelines.components.transformers.encoders.onehot_e method*), 973
describe() (*evalml.pipelines.components.transformers.encoders.OneHotE method*), 988
describe() (*evalml.pipelines.components.transformers.encoders.ordinal_ method*), 978
describe() (*evalml.pipelines.components.transformers.encoders.OrdinalE method*), 992
describe() (*evalml.pipelines.components.transformers.encoders.target_en method*), 982
describe() (*evalml.pipelines.components.transformers.encoders.TargetEn method*), 995
describe() (*evalml.pipelines.components.transformers.feature_selection.f method*), 998
describe() (*evalml.pipelines.components.transformers.feature_selection.F method*), 1018

describe() (evalml.pipelines.components.transformers.feature_selection.RFCLassifierRFE method), 1249
describe() (evalml.pipelines.components.transformers.feature_selection.RFCLassifierSelectFromModel method), 1253
describe() (evalml.pipelines.components.transformers.feature_selection.RFRegressorRFE method), 1256
describe() (evalml.pipelines.components.transformers.feature_selection.RFRegressorSelectFromModel method), 1259
describe() (evalml.pipelines.components.transformers.imputers.PerColumnImputer method), 1261
describe() (evalml.pipelines.components.transformers.PolynomialDecomposer method), 1264
describe() (evalml.pipelines.components.transformers.preprocessing.DateDecomposer method), 1267
describe() (evalml.pipelines.components.transformers.preprocessing.DateTimeFeaturizer method), 1069
describe() (evalml.pipelines.components.transformers.preprocessing.DelayedFeatureTransformer method), 1129
describe() (evalml.pipelines.components.transformers.preprocessing.DelayedFeatureTransformer method), 1133
describe() (evalml.pipelines.components.transformers.preprocessing.decomposer method), 1073
describe() (evalml.pipelines.components.transformers.preprocessing.DFSTransformer method), 1137
describe() (evalml.pipelines.components.transformers.preprocessing.drop_columns method), 1077
describe() (evalml.pipelines.components.transformers.preprocessing.drop_rows method), 1080
describe() (evalml.pipelines.components.transformers.preprocessing.drop_rows method), 1083
describe() (evalml.pipelines.components.transformers.preprocessing.DropNullColumns method), 1139
describe() (evalml.pipelines.components.transformers.preprocessing.DropRowsOfGivenValue method), 1141
describe() (evalml.pipelines.components.transformers.preprocessing.DropSingleIntegerValueCols method), 1144
describe() (evalml.pipelines.components.transformers.preprocessing.EmailFeaturizer method), 1146
describe() (evalml.pipelines.components.transformers.preprocessing.featuretools.DeprecatedFeatureTransformer method), 1086
describe() (evalml.pipelines.components.transformers.preprocessing.log_transformer method), 1089
describe() (evalml.pipelines.components.transformers.preprocessing.LogTransformer method), 1149
describe() (evalml.pipelines.components.transformers.preprocessing.LSA method), 1151
describe() (evalml.pipelines.components.transformers.preprocessing.lsa.LSA method), 1092
describe() (evalml.pipelines.components.transformers.preprocessing.natural_language.NaturalLanguageFeaturizer method), 1095
describe() (evalml.pipelines.components.transformers.preprocessing.NaturalLanguageFeaturizer method), 1154
describe() (evalml.pipelines.components.transformers.preprocessing.polynomial_decomposer.PolynomialDecomposer method), 1099

describe() (evalml.pipelines.components.transformers.preprocessing.PolynomialDetrender method), 1157
describe() (evalml.pipelines.components.transformers.preprocessing.RemoveNullRowsTransformer method), 1104
describe() (evalml.pipelines.components.transformers.preprocessing.RemoveNullRowsTransformer method), 1161
describe() (evalml.pipelines.components.transformers.preprocessing.STLDecomposer method), 1108
describe() (evalml.pipelines.components.transformers.preprocessing.STLDecomposer method), 1164
describe() (evalml.pipelines.components.transformers.preprocessing.TextFeaturizer method), 1113
describe() (evalml.pipelines.components.transformers.preprocessing.TextFeaturizer method), 1169
describe() (evalml.pipelines.components.transformers.preprocessing.TimeSeriesFeaturizer method), 1117
describe() (evalml.pipelines.components.transformers.preprocessing.TimeSeriesFeaturizer method), 1120
describe() (evalml.pipelines.components.transformers.preprocessing.TimeSeriesFeaturizer method), 1172
describe() (evalml.pipelines.components.transformers.preprocessing.TimeSeriesRegularizer method), 1175
describe() (evalml.pipelines.components.transformers.preprocessing.TimeSeriesRegularizer method), 1123
describe() (evalml.pipelines.components.transformers.preprocessing.URLFeaturizer method), 1126
describe() (evalml.pipelines.components.transformers.preprocessing.URLFeaturizer method), 1178
describe() (evalml.pipelines.components.transformers.ReplaceNullableTypes method), 1272
describe() (evalml.pipelines.components.transformers.RFClassifierRFE method), 1274
describe() (evalml.pipelines.components.transformers.RFClassifierSelectFromModel method), 1277
describe() (evalml.pipelines.components.transformers.RFRegressorRFE method), 1281
describe() (evalml.pipelines.components.transformers.RFRegressorSelectFromModel method), 1283
describe() (evalml.pipelines.components.transformers.samplers.BaseSampler method), 1181
describe() (evalml.pipelines.components.transformers.samplers.Oversampler method), 1191
describe() (evalml.pipelines.components.transformers.samplers.Oversampler method), 1184
describe() (evalml.pipelines.components.transformers.samplers.Undersampler method), 1193
describe() (evalml.pipelines.components.transformers.samplers.Undersampler method), 1188
describe() (evalml.pipelines.components.transformers.scalers.StandardScaler method), 1196
describe() (evalml.pipelines.components.transformers.scalers.StandardScaler method), 1199
describe() (evalml.pipelines.components.transformers.SelectByType method), 1286
describe() (evalml.pipelines.components.transformers.SelectColumns method), 1288
describe() (evalml.pipelines.components.transformers.SimpleImputer method), 1291
describe() (evalml.pipelines.components.transformers.StandardScaler method), 1293
describe() (evalml.pipelines.components.transformers.STLDecomposer method), 1296
describe() (evalml.pipelines.components.transformers.TargetEncoder method), 1301
describe() (evalml.pipelines.components.transformers.TargetImputer method), 1304
describe() (evalml.pipelines.components.transformers.TimeSeriesFeaturizer method), 1307
describe() (evalml.pipelines.components.transformers.TimeSeriesImputer method), 1310
describe() (evalml.pipelines.components.transformers.TimeSeriesRegularizer method), 1313
describe() (evalml.pipelines.components.transformers.Transformer method), 1316
describe() (evalml.pipelines.components.transformers.transformer.Transformer method), 1212
describe() (evalml.pipelines.components.UnstackTwoFeatureUndersampler method), 1319
describe() (evalml.pipelines.components.URLFeaturizer method), 1322
describe() (evalml.pipelines.components.Undersampler method), 1533
describe() (evalml.pipelines.components.URLFeaturizer method), 1536
describe() (evalml.pipelines.components.VowpalWabbitBinaryClassifier method), 1538
describe() (evalml.pipelines.components.VowpalWabbitMulticlassClassifier method), 1541
describe() (evalml.pipelines.components.VowpalWabbitRegressor method), 1544
describe() (evalml.pipelines.components.XGBoostClassifier method), 1548
describe() (evalml.pipelines.components.XGBoostRegressor method), 1551
describe() (evalml.pipelines.DecisionTreeClassifier method), 1672
describe() (evalml.pipelines.DecisionTreeRegressor method), 1676
describe() (evalml.pipelines.DFSTransformer method), 1679
describe() (evalml.pipelines.DropNaNRowsTransformer method), 1682
describe() (evalml.pipelines.ElasticNetClassifier method), 1685
describe() (evalml.pipelines.ElasticNetRegressor method), 1688
describe() (evalml.pipelines.Estimator method), 1691
describe() (evalml.pipelines.ExponentialSmoothingRegressor

*method*), 1695
describe() (*evalml.pipelines.ExtraTreesClassifier method*), 1698
describe() (*evalml.pipelines.ExtraTreesRegressor method*), 1702
describe() (*evalml.pipelines.FeatureSelector method*), 1705
describe() (*evalml.pipelines.Imputer method*), 1708
describe() (*evalml.pipelines.KNeighborsClassifier method*), 1711
describe() (*evalml.pipelines.LightGBMClassifier method*), 1715
describe() (*evalml.pipelines.LightGBMRegressor method*), 1718
describe() (*evalml.pipelines.LinearRegressor method*), 1721
describe() (*evalml.pipelines.LogisticRegressionClassifier method*), 1724
describe() (*evalml.pipelines.multiclass_classification_pipelines.MulticlassClassificationPipeline method*), 1577
describe() (*evalml.pipelines.MulticlassClassificationPipeline method*), 1729
describe() (*evalml.pipelines.OneHotEncoder method*), 1735
describe() (*evalml.pipelines.OrdinalEncoder method*), 1738
describe() (*evalml.pipelines.PerColumnImputer method*), 1740
describe() (*evalml.pipelines.pipeline_base.PipelineBase method*), 1584
describe() (*evalml.pipelines.PipelineBase method*), 1744
describe() (*evalml.pipelines.ProphetRegressor method*), 1749
describe() (*evalml.pipelines.RandomForestClassifier method*), 1752
describe() (*evalml.pipelines.RandomForestRegressor method*), 1755
describe() (*evalml.pipelines.regression_pipeline.RegressionPipeline method*), 1592
describe() (*evalml.pipelines.RegressionPipeline method*), 1760
describe() (*evalml.pipelines.RFClassifierSelectFromModel method*), 1765
describe() (*evalml.pipelines.RFRegressorSelectFromModel method*), 1768
describe() (*evalml.pipelines.SimpleImputer method*), 1770
describe() (*evalml.pipelines.StackedEnsembleBase method*), 1773
describe() (*evalml.pipelines.StackedEnsembleClassifier method*), 1777
describe() (*evalml.pipelines.StackedEnsembleRegressor method*), 1781
describe() (*evalml.pipelines.StandardScaler method*), 1783
describe() (*evalml.pipelines.SVMClassifier method*), 1786
describe() (*evalml.pipelines.SVMRegressor method*), 1789
describe() (*evalml.pipelines.TargetEncoder method*), 1792
describe() (*evalml.pipelines.time_series_classification_pipelines.TimeSe method*), 1599
describe() (*evalml.pipelines.time_series_classification_pipelines.TimeSe method*), 1606
describe() (*evalml.pipelines.time_series_classification_pipelines.TimeSe method*), 1614
describe() (*evalml.pipelines.time_series_pipeline_base.TimeSeriesPipeli method*), 1622
describe() (*evalml.pipelines.time_series_regression_pipeline.TimeSeries method*), 1630
describe() (*evalml.pipelines.TimeSeriesBinaryClassificationPipeline method*), 1797
describe() (*evalml.pipelines.TimeSeriesClassificationPipeline method*), 1804
describe() (*evalml.pipelines.TimeSeriesFeaturizer method*), 1811
describe() (*evalml.pipelines.TimeSeriesImputer method*), 1813
describe() (*evalml.pipelines.TimeSeriesMulticlassClassificationPipeline method*), 1818
describe() (*evalml.pipelines.TimeSeriesRegressionPipeline method*), 1825
describe() (*evalml.pipelines.TimeSeriesRegularizer method*), 1833
describe() (*evalml.pipelines.Transformer method*), 1835
describe() (*evalml.pipelines.VowpalWabbitBinaryClassifier method*), 1838
describe() (*evalml.pipelines.VowpalWabbitMulticlassClassifier method*), 1841
describe() (*evalml.pipelines.VowpalWabbitRegressor method*), 1844
describe() (*evalml.pipelines.XGBoostClassifier method*), 1847
describe() (*evalml.pipelines.XGBoostRegressor method*), 1850
describe_pipeline() (*evalml.automl.automl_search.AutoMLSearch method*), 261
describe_pipeline() (*evalml.automl.AutoMLSearch method*), 277
describe_pipeline() (*evalml.AutoMLSearch method*), 1917
detect_problem_type() (*in module evalml.problem_types*), 1880
detect_problem_type() (*in module*

*evalml.problem_types.utils*), 1877
determine_periodicity() (*evalml.pipelines.components.PolynomialDecomposer class method*), 1452
determine_periodicity() (*evalml.pipelines.components.STLDecomposer class method*), 1501
determine_periodicity() (*evalml.pipelines.components.transformers.PolynomialDecomposer class method*), 1267
determine_periodicity() (*evalml.pipelines.components.transformers.preprocessing.DecomposerBase class method*), 1133
determine_periodicity() (*evalml.pipelines.components.transformers.preprocessing.decomposer.PolynomialDecomposer class method*), 1073
determine_periodicity() (*evalml.pipelines.components.transformers.preprocessing.polynomial_decomposer.PolynomialDecomposer class method*), 1099
determine_periodicity() (*evalml.pipelines.components.transformers.preprocessing.STLDecomposer class method*), 1157
determine_periodicity() (*evalml.pipelines.components.transformers.preprocessing.stl_decomposer.STLDecomposer class method*), 1108
determine_periodicity() (*evalml.pipelines.components.transformers.preprocessing.time_series_base_decomposer.TimeSeriesBaseDecomposer class method*), 1164
determine_periodicity() (*evalml.pipelines.components.transformers.STLDecomposer class method*), 1296
DFSTransformer (*class in evalml.pipelines*), 1678
DFSTransformer (*class in evalml.pipelines.components*), 1366
DFSTransformer (*class in evalml.pipelines.components.transformers*), 1218
DFSTransformer (*class in evalml.pipelines.components.transformers.preprocessing*), 1135
DFSTransformer (*class in evalml.pipelines.components.transformers.preprocessing.featureselection*), 1085
done() (*evalml.automl.engine.cf_engine.CFComputation method*), 239
done() (*evalml.automl.engine.dask_engine.DaskComputation method*), 242
done() (*evalml.automl.engine.engine_base.EngineComputation method*), 245
done() (*evalml.automl.engine.EngineComputation method*), 254
done() (*evalml.automl.engine.sequential_engine.SequentialComputation method*), 248
downcast_nullable_types() (*in module evalml.utils*), 1910
downcast_nullable_types() (*in module evalml.utils.woodwork_utils*), 1908
DOWNCAST_TYPE_DICT (*in module evalml.utils.nullable_type_utils*), 1907
drop_rows_with_nans() (*in module evalml.utils*), 1911
drop_rows_with_nans() (*in module evalml.utils.gen_utils*), 1903
DropColumns (*class in evalml.pipelines.components*), 1369
DropColumns (*class in evalml.pipelines.components.transformers*), 1221
DropColumns (*class in evalml.pipelines.components.transformers.column_selectors*), 1204
DropNaNRowsTransformer (*class in evalml.pipelines*), 1681
DropNaNRowsTransformer (*class in evalml.pipelines.components*), 1371
DropNaNRowsTransformer (*class in evalml.pipelines.components.transformers*), 1223
DropNaNRowsTransformer (*class in evalml.pipelines.components.transformers.preprocessing*), 1138
DropNaNRowsTransformer (*class in evalml.pipelines.components.transformers.preprocessing.drop_na*), 1076
DropNullColumns (*class in evalml.pipelines.components*), 1374
DropNullColumns (*class in evalml.pipelines.components.transformers*), 1226
DropNullColumns (*class in evalml.pipelines.components.transformers.preprocessing*), 1140
DropNullColumns (*class in evalml.pipelines.components.transformers.preprocessing.drop_nu*), 1079
DropRowsTransformer (*class in evalml.pipelines.components*), 1376
DropRowsTransformer (*class in evalml.pipelines.components.transformers*), 1228
DropRowsTransformer (*class in evalml.pipelines.components.transformers.preprocessing*), 1143
DropRowsTransformer (*class in evalml.pipelines.components.transformers.preprocessing.drop_ro*), 1082


## E


elapsed() (*evalml.automl.Progress method*), 281

elapsed() (*evalml.automl.progress.Progress method*), 268
ElasticNetClassifier (*class in evalml.pipelines*), 1683
ElasticNetClassifier (*class in evalml.pipelines.components*), 1379
ElasticNetClassifier (*class in evalml.pipelines.components.estimators*), 883
ElasticNetClassifier (*class in evalml.pipelines.components.estimators.classifiers*), 706
ElasticNetClassifier (*class in evalml.pipelines.components.estimators.classifiers.elastic_net_classifier*), 654
ElasticNetRegressor (*class in evalml.pipelines*), 1687
ElasticNetRegressor (*class in evalml.pipelines.components*), 1382
ElasticNetRegressor (*class in evalml.pipelines.components.estimators*), 886
ElasticNetRegressor (*class in evalml.pipelines.components.estimators.regressors*), 816
ElasticNetRegressor (*class in evalml.pipelines.components.estimators.regressors.elastic_net_regressor*), 758
EmailFeaturizer (*class in evalml.pipelines.components*), 1385
EmailFeaturizer (*class in evalml.pipelines.components.transformers*), 1230
EmailFeaturizer (*class in evalml.pipelines.components.transformers.preprocessing*), 1145
EmailFeaturizer (*class in evalml.pipelines.components.transformers.preprocessing.text_featurizer*), 1123
EngineBase (*class in evalml.automl*), 279
EngineBase (*class in evalml.automl.engine*), 254
EngineBase (*class in evalml.automl.engine.engine_base*), 245
EngineComputation (*class in evalml.automl.engine*), 254
EngineComputation (*class in evalml.automl.engine.engine_base*), 245
error() (*evalml.automl.engine.engine_base.JobLogger method*), 246
Estimator (*class in evalml.pipelines*), 1690
Estimator (*class in evalml.pipelines.components*), 1388
Estimator (*class in evalml.pipelines.components.estimators*), 889
Estimator (*class in evalml.pipelines.components.estimators.estimator*), 854
estimator_unable_to_handle_nans() (*in module evalml.pipelines.components.utils*), 1328
evalml
module, 223
evalml.automl
module, 223
evalml.automl.automl_algorithm
module, 223
evalml.automl.automl_algorithm.automl_algorithm
module, 223
evalml.automl.automl_algorithm.default_algorithm
module, 225
evalml.automl.automl_algorithm.iterative_algorithm
module, 228
evalml.automl.automl_search
module, 257
evalml.automl.callbacks
module, 266
evalml.automl.engine
module, 238
evalml.automl.engine.cf_engine
module, 238
evalml.automl.engine.dask_engine
module, 241
evalml.automl.engine.engine_base
module, 244
evalml.automl.engine.sequential_engine
module, 248
evalml.automl.pipeline_search_plots
module, 267
evalml.automl.progress
module, 268
evalml.automl.utils
module, 269
evalml.data_checks
module, 285
evalml.data_checks.class_imbalance_data_check
module, 286
evalml.data_checks.data_check
module, 289
evalml.data_checks.data_check_action
module, 289
evalml.data_checks.data_check_action_code
module, 290
evalml.data_checks.data_check_action_option
module, 291
evalml.data_checks.data_check_message
module, 294
evalml.data_checks.data_check_message_code
module, 296
evalml.data_checks.data_check_message_type
module, 298
evalml.data_checks.data_checks
module, 299

evalml.data_checks.datetime_format_data_check
module, 299
evalml.data_checks.default_data_checks
module, 307
evalml.data_checks.id_columns_data_check
module, 308
evalml.data_checks.invalid_target_data_check
module, 312
evalml.data_checks.multicollinearity_data_check
module, 316
evalml.data_checks.no_variance_data_check
module, 318
evalml.data_checks.null_data_check
module, 321
evalml.data_checks.outliers_data_check
module, 325
evalml.data_checks.sparsity_data_check
module, 328
evalml.data_checks.target_distribution_data_check
module, 330
evalml.data_checks.target_leakage_data_check
module, 332
evalml.data_checks.ts_parameters_data_check
module, 334
evalml.data_checks.ts_splitting_data_check
module, 336
evalml.data_checks.uniqueness_data_check
module, 338
evalml.data_checks.utils
module, 340
evalml.demos
module, 387
evalml.demos.breast_cancer
module, 387
evalml.demos.churn
module, 388
evalml.demos.diabetes
module, 388
evalml.demos.fraud
module, 389
evalml.demos.weather
module, 389
evalml.demos.wine
module, 390
evalml.exceptions
module, 392
evalml.exceptions.exceptions
module, 392
evalml.model_family
module, 398
evalml.model_family.model_family
module, 398
evalml.model_family.utils
module, 399
evalml.model_understanding
module, 402
evalml.model_understanding.decision_boundary
module, 409
evalml.model_understanding.feature_explanations
module, 410
evalml.model_understanding.force_plots
module, 411
evalml.model_understanding.metrics
module, 413
evalml.model_understanding.partial_dependence_functions
module, 416
evalml.model_understanding.permutation_importance
module, 418
evalml.model_understanding.prediction_explanations
module, 402
evalml.model_understanding.prediction_explanations.explain
module, 402
evalml.model_understanding.visualizations
module, 420
evalml.objectives
module, 440
evalml.objectives.binary_classification_objective
module, 440
evalml.objectives.cost_benefit_matrix
module, 443
evalml.objectives.fraud_cost
module, 446
evalml.objectives.lead_scoring
module, 449
evalml.objectives.multiclass_classification_objective
module, 452
evalml.objectives.objective_base
module, 454
evalml.objectives.regression_objective
module, 457
evalml.objectives.sensitivity_low_alert
module, 459
evalml.objectives.standard_metrics
module, 462
evalml.objectives.time_series_regression_objective
module, 527
evalml.objectives.utils
module, 530
evalml.pipelines
module, 619
evalml.pipelines.binary_classification_pipeline
module, 1552
evalml.pipelines.binary_classification_pipeline_mixin
module, 1560
evalml.pipelines.classification_pipeline
module, 1561
evalml.pipelines.component_graph
module, 1568

evalml.pipelines.components
module, 619
evalml.pipelines.components.component_base
module, 1323
evalml.pipelines.components.component_base_meta
module, 1326
evalml.pipelines.components.ensemble
module, 619
evalml.pipelines.components.ensemble.stacked_ensemble_base
module, 619
evalml.pipelines.components.ensemble.stacked_ensemble_classifier
module, 622
evalml.pipelines.components.ensemble.stacked_ensemble_regressor
module, 627
evalml.pipelines.components.estimators
module, 642
evalml.pipelines.components.estimators.classifiers
module, 642
evalml.pipelines.components.estimators.classifiers.baseline_classifier
module, 642
evalml.pipelines.components.estimators.classifiers.catboost_classifier
module, 646
evalml.pipelines.components.estimators.classifiers.decision_tree_classifier
module, 650
evalml.pipelines.components.estimators.classifiers.elastic_net_classifier
module, 654
evalml.pipelines.components.estimators.classifiers.extra_trees_classifier
module, 658
evalml.pipelines.components.estimators.classifiers.k_neighbors_classifier
module, 663
evalml.pipelines.components.estimators.classifiers.lightgbm_classifier
module, 667
evalml.pipelines.components.estimators.classifiers.logistic_regression_classifier
module, 671
evalml.pipelines.components.estimators.classifiers.rf_classifier
module, 675
evalml.pipelines.components.estimators.classifiers.svm_classifier
module, 678
evalml.pipelines.components.estimators.classifiers.vowpal_wabbit_classifiers
module, 682
evalml.pipelines.components.estimators.classifiers.xgboost_classifier
module, 692
evalml.pipelines.components.estimators.estimator
module, 853
evalml.pipelines.components.estimators.regressors
module, 741
evalml.pipelines.components.estimators.regressors.baseline_regressor
module, 741
evalml.pipelines.components.estimators.regressors.elastic_net_regressor
module, 746
evalml.pipelines.components.estimators.regressors.extra_trees_regressor
module, 749
evalml.pipelines.components.estimators.regressors.decision_tree_regressor
module, 753
evalml.pipelines.components.estimators.regressors.elastic
module, 758
evalml.pipelines.components.estimators.regressors.et_regre
module, 761
evalml.pipelines.components.estimators.regressors.exponent
module, 766
evalml.pipelines.components.estimators.regressors.lightgbm
module, 770
evalml.pipelines.components.estimators.regressors.linear_r
module, 774
evalml.pipelines.components.estimators.regressors.prophet_
module, 777
evalml.pipelines.components.estimators.regressors.rf_regre
module, 782
evalml.pipelines.components.estimators.regressors.svm_regr
module, 786
evalml.pipelines.components.estimators.regressors.time_ser
module, 790
evalml.pipelines.components.estimators.regressors.vowpal_w
module, 793
evalml.pipelines.components.estimators.regressors.xgboost_
module, 797
evalml.pipelines.components.transformers
module, 956
evalml.pipelines.components.transformers.column_selectors
module, 1201
evalml.pipelines.components.transformers.dimensionality_re
module, 957
evalml.pipelines.components.transformers.dimensionality_re
module, 957
evalml.pipelines.components.transformers.dimensionality_re
module, 960
evalml.pipelines.components.transformers.encoders
module, 968
evalml.pipelines.components.transformers.encoders.label_en
module, 968
evalml.pipelines.components.transformers.encoders.onehot_e
module, 971
evalml.pipelines.components.transformers.encoders.ordinal_
module, 976
evalml.pipelines.components.transformers.encoders.target_e
module, 980
evalml.pipelines.components.transformers.feature_selection
module, 996
evalml.pipelines.components.transformers.feature_selection
module, 996
evalml.pipelines.components.transformers.feature_selection
module, 999
evalml.pipelines.components.transformers.feature_selection
module, 1009
evalml.pipelines.components.transformers.feature_selection
module, 1013
evalml.pipelines.components.transformers.imputers
module, 1032

evalml.pipelines.components.transformers.imputers.imputer
module, 1032
evalml.pipelines.components.transformers.imputers.knn_imputer
module, 1036
evalml.pipelines.components.transformers.imputers.per_column_imputer
module, 1039
evalml.pipelines.components.transformers.imputers.simple_imputer
module, 1042
evalml.pipelines.components.transformers.imputers.target_imputer
module, 1045
evalml.pipelines.components.transformers.imputers.time_series_imputer
module, 1048
evalml.pipelines.components.transformers.preprocessing
module, 1068
evalml.pipelines.components.transformers.preprocessing.delete_rows_transformer
module, 1068
evalml.pipelines.components.transformers.preprocessing.drop_columns_transformer
module, 1071
evalml.pipelines.components.transformers.preprocessing.drop_rows_transformer
module, 1076
evalml.pipelines.components.transformers.preprocessing.lagged_feature_extractor
module, 1079
evalml.pipelines.components.transformers.preprocessing.polynomial_decomposer
module, 1082
evalml.pipelines.components.transformers.preprocessing.text_featurizer
module, 1085
evalml.pipelines.components.transformers.preprocessing.time_series_featurizer
module, 1088
evalml.pipelines.components.transformers.preprocessing.time_series_ordinalizer
module, 1091
evalml.pipelines.components.transformers.preprocessing.time_series_rolling_transformer
module, 1094
evalml.pipelines.components.transformers.preprocessing.time_series_target_encoder
module, 1097
evalml.pipelines.components.transformers.preprocessing.time_series_utilities
module, 1103
evalml.pipelines.components.transformers.preprocessing.utils
module, 1106
evalml.pipelines.components.transformers.preprocessing.delay_target_transformer
module, 1112
evalml.pipelines.components.transformers.preprocessing.drop_nan_target_rows_transformer
module, 1115
evalml.pipelines.components.transformers.preprocessing.drop_rows_transformer
module, 1119
evalml.pipelines.components.transformers.preprocessing.transformer
module, 1122
evalml.pipelines.components.transformers.samplers
module, 1180
evalml.pipelines.components.transformers.samplers.base_sampler
module, 1180
evalml.pipelines.components.transformers.samplers.oversamplers
module, 1183
evalml.pipelines.components.transformers.samplers.undersampler
module, 1186
evalml.pipelines.components.transformers.scalers
module, 1195
evalml.pipelines.components.transformers.scalers.standard_scaler
module, 1195
evalml.pipelines.components.transformers.transformer
module, 1211
evalml.pipelines.components.utils
module, 1327
evalml.pipelines.multiclass_classification_pipeline
module, 1574
evalml.pipelines.pipeline_base
module, 1581
evalml.pipelines.pipeline_meta
module, 1587
evalml.pipelines.regression_pipeline
module, 1589
evalml.pipelines.time_series_classification_pipelines
module, 1595
evalml.pipelines.time_series_transformer_pipeline_base
module, 1619
evalml.pipelines.time_series_regression_pipeline
module, 1626
evalml.pipelines.utils
module, 1635
evalml.preprocessing
module, 1852
evalml.preprocessing.data_splitters
module, 1852
evalml.preprocessing.data_splitters.no_split
module, 1853
evalml.preprocessing.data_splitters.rolling_origin_cv
module, 1854
evalml.preprocessing.data_splitters.time_series_split
module, 1856
evalml.preprocessing.data_splitters.training_validation_split
module, 1858
evalml.preprocessing.utils
module, 1866
evalml.problem_types
module, 1875
evalml.problem_types.feature_types
module, 1876
evalml.problem_types.regularizer
module, 1877
evalml.utils.transform_primitive_components
module, 1883
evalml.tuners.grid_search_tuner
module, 1883
evalml.tuners.random_search_tuner
module, 1885
evalml.tuners.skopt_tuner
module, 1887
evalml.tuners.tuner
module, 1889

evalml.tuners.tuner_exceptions
module, 1891
evalml.utils
module, 1897
evalml.utils.base_meta
module, 1897
evalml.utils.cli_utils
module, 1898
evalml.utils.gen_utils
module, 1900
evalml.utils.logger
module, 1906
evalml.utils.nullable_type_utils
module, 1907
evalml.utils.update_checker
module, 1907
evalml.utils.woodwork_utils
module, 1907
evaluate_pipeline() (*in module evalml.automl.engine*), 254
evaluate_pipeline() (*in module evalml.automl.engine.engine_base*), 245
expected_range (*evalml.objectives.binary_classification_objective.BinaryClassificationObjective property*), 442
expected_range (*evalml.objectives.BinaryClassificationObjective property*), 551
expected_range (*evalml.objectives.multiclass_classification_objective.MulticlassClassificationObjective property*), 453
expected_range (*evalml.objectives.MulticlassClassificationObjective property*), 591
expected_range (*evalml.objectives.objective_base.ObjectiveBase property*), 455
expected_range (*evalml.objectives.ObjectiveBase property*), 593
expected_range (*evalml.objectives.regression_objective.RegressionObjective property*), 458
expected_range (*evalml.objectives.RegressionObjective property*), 612
expected_range (*evalml.objectives.time_series_regression_objective.TimeSeriesRegressionObjective property*), 529
explain_predictions() (*in module evalml.model_understanding*), 429
explain_predictions() (*in module evalml.model_understanding.prediction_explanations*), 406
explain_predictions() (*in module evalml.model_understanding.prediction_explanations.explainers*), 403
explain_predictions_best_worst() (*in module evalml.model_understanding*), 430
explain_predictions_best_worst() (*in module evalml.model_understanding.prediction_explanations*), 407
explain_predictions_best_worst() (*in module evalml.model_understanding.prediction_explanations.explainers*), 404
ExplainPredictionsStage (*class in evalml.model_understanding.prediction_explanations.explainers*), 405
ExponentialSmoothingRegressor (*class in evalml.pipelines*), 1693
ExponentialSmoothingRegressor (*class in evalml.pipelines.components*), 1391
ExponentialSmoothingRegressor (*class in evalml.pipelines.components.estimators*), 893
ExponentialSmoothingRegressor (*class in evalml.pipelines.components.estimators.regressors*), 819
ExponentialSmoothingRegressor (*class in evalml.pipelines.components.estimators.regressors.exponential_s*), 766
ExpVariance (*class in evalml.objectives*), 554
ExpVariance (*class in evalml.objectives.standard_metrics*), 478
ExtraTreesClassifier (*class in evalml.pipelines*), 1696
ExtraTreesClassifier (*class in evalml.pipelines.components*), 1394
ExtraTreesClassifier (*class in evalml.pipelines.components.estimators*), 896
ExtraTreesClassifier (*class in evalml.pipelines.components.estimators.classifiers*), 709
ExtraTreesClassifier (*class in evalml.pipelines.components.estimators.classifiers.et_classifier*), 659
ExtraTreesRegressor (*class in evalml.pipelines*), 1700
ExtraTreesRegressor (*class in evalml.pipelines.components*), 1398
ExtraTreesRegressor (*class in evalml.pipelines.components.estimators*), 900
ExtraTreesRegressor (*class in evalml.pipelines.components.estimators.regressors*), 822
ExtraTreesRegressor (*class in evalml.pipelines.components.estimators.regressors.et_regressor*), 761


## F


F1 (*class in evalml.objectives*), 556
F1 (*class in evalml.objectives.standard_metrics*), 480
F1Macro (*class in evalml.objectives*), 558
F1Macro (*class in evalml.objectives.standard_metrics*), 482
F1Micro (*class in evalml.objectives*), 560

F1Micro (*class in evalml.objectives.standard_metrics*), 484
F1Weighted (*class in evalml.objectives*), 562
F1Weighted (*class in evalml.objectives.standard_metrics*), 486
feature_importance (*evalml.pipelines.ARIMARegressor property*), 1644
feature_importance (*evalml.pipelines.binary_classification_pipeline.BinaryClassificationPipeline property*), 1556
feature_importance (*evalml.pipelines.BinaryClassificationPipeline property*), 1648
feature_importance (*evalml.pipelines.CatBoostClassifier property*), 1654
feature_importance (*evalml.pipelines.CatBoostRegressor property*), 1657
feature_importance (*evalml.pipelines.classification_pipeline.ClassificationPipeline property*), 1564
feature_importance (*evalml.pipelines.ClassificationPipeline property*), 1661
feature_importance (*evalml.pipelines.components.ARIMARegressor property*), 1339
feature_importance (*evalml.pipelines.components.BaselineClassifier property*), 1342
feature_importance (*evalml.pipelines.components.BaselineRegressor property*), 1345
feature_importance (*evalml.pipelines.components.CatBoostClassifier property*), 1348
feature_importance (*evalml.pipelines.components.CatBoostRegressor property*), 1352
feature_importance (*evalml.pipelines.components.DecisionTreeClassifier property*), 1361
feature_importance (*evalml.pipelines.components.DecisionTreeRegressor property*), 1365
feature_importance (*evalml.pipelines.components.ElasticNetClassifier property*), 1381
feature_importance (*evalml.pipelines.components.ElasticNetRegressor property*), 1384
feature_importance (*evalml.pipelines.components.ensemble.StackedEnsembleBase property*), 621
feature_importance (*evalml.pipelines.components.ensemble.StackedEnsembleClassifier property*), 625
feature_importance (*evalml.pipelines.components.ensemble.StackedEnsembleRegressor property*), 629
feature_importance (*evalml.pipelines.components.ensemble.TimeSeriesEnsemble property*), 632
feature_importance (*evalml.pipelines.components.ensemble.TimeSeriesEnsemble.TimeSeriesEnsembleClassifier property*), 636
feature_importance (*evalml.pipelines.components.ensemble.TimeSeriesEnsemble.TimeSeriesEnsembleRegressor property*), 640
feature_importance (*evalml.pipelines.components.Estimator property*), 1389
feature_importance (*evalml.pipelines.components.estimators.ARIMARegressor property*), 861
feature_importance (*evalml.pipelines.components.estimators.BaselineClassifier property*), 864
feature_importance (*evalml.pipelines.components.estimators.BaselineRegressor property*), 867
feature_importance (*evalml.pipelines.components.estimators.CatBoostClassifier property*), 870
feature_importance (*evalml.pipelines.components.estimators.CatBoostRegressor property*), 874
feature_importance (*evalml.pipelines.components.estimators.classifiers.DecisionTreeClassifier property*), 644
feature_importance (*evalml.pipelines.components.estimators.classifiers.ElasticNetClassifier property*), 698
feature_importance (*evalml.pipelines.components.estimators.classifiers.ExtraTreesClassifier property*), 648
feature_importance (*evalml.pipelines.components.estimators.classifiers.KNeighborsClassifier property*), 701
feature_importance (*evalml.pipelines.components.estimators.classifiers.LightGBMClassifier property*), 652
feature_importance (*evalml.pipelines.components.estimators.classifiers.LinearDiscriminantAnalysis property*), 704
feature_importance (*evalml.pipelines.components.estimators.classifiers.LogisticRegressionClassifier property*), 657
feature_importance (*evalml.pipelines.components.estimators.classifiers.RandomForestClassifier property*), 708
feature_importance (*evalml.pipelines.components.estimators.classifiers.RandomForestRegressor property*), 661
feature_importance (*evalml.pipelines.components.estimators.classifiers.StackedEnsembleClassifier property*), 712
feature_importance (*evalml.pipelines.components.estimators.classifiers.StackedEnsembleRegressor property*), 665
feature_importance (*evalml.pipelines.components.estimators.classifiers.TimeSeriesClassifierEstimator property*), 715
feature_importance (*evalml.pipelines.components.estimators.classifiers.TimeSeriesRegressorEstimator property*), 669
feature_importance (*evalml.pipelines.components.estimators.classifiers.VowpalWabbitBinaryClassifier property*), 719
feature_importance (*evalml.pipelines.components.estimators.classifiers.VowpalWabbitMulticlassClassifier property*), 673
feature_importance (*evalml.pipelines.components.estimators.classifiers.XGBoostClassifier property*), 722
feature_importance (*evalml.pipelines.components.estimators.classifiers.XGBoostRFClassifier property*), 726
feature_importance (*evalml.pipelines.components.estimators.classifiers.XGBoostRFRegressor property*), 676
feature_importance (*evalml.pipelines.components.estimators.regressors.DecisionTreeRegressor property*), 680
feature_importance (*evalml.pipelines.components.estimators.regressors.ElasticNetRegressor property*), 729
feature_importance (*evalml.pipelines.components.estimators.regressors.ExtraTreesRegressor property*), 684
feature_importance (*evalml.pipelines.components.estimators.regressors.KNeighborsRegressor property*), 687
feature_importance (*evalml.pipelines.components.estimators.regressors.LightGBMRegressor property*), 690
feature_importance (*evalml.pipelines.components.estimators.regressors.LinearRegressor*

*property*), 732
feature_importance(*evalml.pipelines.components.estimators.classifiers.VowpalWabbitMulticlassClassifier*
*property*), 736
feature_importance(*evalml.pipelines.components.estimators.classifiers.XGBoostClassifier*
*property*), 694
feature_importance(*evalml.pipelines.components.estimators.classifiers.XGBoostRFClassifier*
*property*), 739
feature_importance(*evalml.pipelines.components.estimators.classifiers.DecisionTreeClassifier*
*property*), 877
feature_importance(*evalml.pipelines.components.estimators.classifiers.DecisionTreeRanger*
*property*), 881
feature_importance(*evalml.pipelines.components.estimators.classifiers.ElasticNetClassifier*
*property*), 885
feature_importance(*evalml.pipelines.components.estimators.classifiers.ElasticNetRanger*
*property*), 888
feature_importance(*evalml.pipelines.components.estimators.classifiers.ExtraTrees*
*property*), 891
feature_importance(*evalml.pipelines.components.estimators.classifiers.ExtraTreesClassifier*
*property*), 855
feature_importance(*evalml.pipelines.components.estimators.classifiers.ExtraTreesRanger*
*property*), 895
feature_importance(*evalml.pipelines.components.estimators.classifiers.ExtreeTreesClassifier*
*property*), 898
feature_importance(*evalml.pipelines.components.estimators.classifiers.ExtreeTreesRanger*
*property*), 902
feature_importance(*evalml.pipelines.components.estimators.classifiers.KNeighborsClassifier*
*property*), 906
feature_importance(*evalml.pipelines.components.estimators.classifiers.LightGBMClassifier*
*property*), 910
feature_importance(*evalml.pipelines.components.estimators.classifiers.LightGBMRanger*
*property*), 913
feature_importance(*evalml.pipelines.components.estimators.classifiers.LinearRanger*
*property*), 916
feature_importance(*evalml.pipelines.components.estimators.classifiers.LogisticRegressionClassifier*
*property*), 920
feature_importance(*evalml.pipelines.components.estimators.classifiers.PerceptronRanger*
*property*), 923
feature_importance(*evalml.pipelines.components.estimators.classifiers.RandomForestClassifier*
*property*), 926
feature_importance(*evalml.pipelines.components.estimators.classifiers.RandomForestRanger*
*property*), 930
feature_importance(*evalml.pipelines.components.estimators.regressors.ARIMARegressor*
*property*), 744
feature_importance(*evalml.pipelines.components.estimators.regressors.ARIMARegressor*
*property*), 804
feature_importance(*evalml.pipelines.components.estimators.regressors.BaselineRegressor*
*property*), 748
feature_importance(*evalml.pipelines.components.estimators.regressors.CatBoostRegressor*
*property*), 807
feature_importance(*evalml.pipelines.components.estimators.regressors.CatBoostRegressor*
*property*), 752
feature_importance(*evalml.pipelines.components.estimators.regressors.DecisionTreeRegressor*
*property*), 810
feature_importance(*evalml.pipelines.components.estimators.regressors.DecisionTreeRegressor*
*property*), 756
feature_importance(*evalml.pipelines.components.estimators.regressors.ElasticNetRegressor*
*property*), 814
feature_importance(*evalml.pipelines.components.estimators.regressors.ElasticNetRegressor*
*property*), 759
feature_importance(*evalml.pipelines.components.estimators.regressors.ExtraTreesRegressor*
*property*), 817
feature_importance(*evalml.pipelines.components.estimators.regressors.ExtraTreesRegressor*
*property*), 764
feature_importance(*evalml.pipelines.components.estimators.regressors.ExtreeTreesRegressor*
*property*), 768
feature_importance(*evalml.pipelines.components.estimators.regressors.KNeighborsRegressor*
*property*), 821
feature_importance(*evalml.pipelines.components.estimators.regressors.LightGBMRegressor*
*property*), 825
feature_importance(*evalml.pipelines.components.estimators.regressors.LinearRegressor*
*property*), 772
feature_importance(*evalml.pipelines.components.estimators.regressors.PerceptronRegressor*
*property*), 828
feature_importance(*evalml.pipelines.components.estimators.regressors.RandomForestRegressor*
*property*), 775
feature_importance(*evalml.pipelines.components.estimators.regressors.RandomForestRegressor*
*property*), 831
feature_importance(*evalml.pipelines.components.estimators.regressors.XGBoostRegressor*
*property*), 780
feature_importance(*evalml.pipelines.components.estimators.regressors.XGBoostRFRegressor*
*property*), 835
feature_importance(*evalml.pipelines.components.estimators.regressors.VowpalWabbitMulticlassClassifier*
*property*), 839
feature_importance(*evalml.pipelines.components.estimators.regressors.VowpalWabbitRegressor*
*property*), 784
feature_importance(*evalml.pipelines.components.estimators.regressors.VowpalWabbitRegressor*
*property*), 788
feature_importance(*evalml.pipelines.components.estimators.regressors.VowpalWabbitRegressorClassifier*
*property*), 842
feature_importance(*evalml.pipelines.components.estimators.regressors.VowpalWabbitRegressor*
*property*), 791
feature_importance(*evalml.pipelines.components.estimators.regressors.VowpalWabbitRegressorClassifier*
*property*), 845
feature_importance(*evalml.pipelines.components.estimators.regressors.VowpalWabbitRegressor*
*property*), 795
feature_importance(*evalml.pipelines.components.estimators.regressors.ARIMARegressor*
*property*), 848
feature_importance(*evalml.pipelines.components.estimators.regressors.ARIMARegressor*
*property*), 799
feature_importance(*evalml.pipelines.components.estimators.regressors.BaselineRegressor*
*property*), 852
feature_importance(*evalml.pipelines.components.estimators.SVMClassifier*
*property*), 933
feature_importance(*evalml.pipelines.components.estimators.SVMRegressor*
*property*), 936
feature_importance(*evalml.pipelines.components.estimators.TimeSeries*
*property*), 939
feature_importance(*evalml.pipelines.components.estimators.VowpalWa*

property), 942
feature_importance(evalml.pipelines.components.estimators.VowpalWabbitMulticlassClassifier
property), 946
feature_importance(evalml.pipelines.components.estimators.VowpalWabbitRegressor
property), 949
feature_importance(evalml.pipelines.components.estimators.XGBoostClassifier
property), 952
feature_importance(evalml.pipelines.components.estimators.XGBoostRegressor
property), 955
feature_importance(evalml.pipelines.components.ExponentialSmoothingRegressor
property), 1393
feature_importance(evalml.pipelines.components.ExtraTreesClassifier
property), 1396
feature_importance(evalml.pipelines.components.ExtraTreesRegressor
property), 1400
feature_importance(evalml.pipelines.components.KNeighborsClassifier
property), 1409
feature_importance(evalml.pipelines.components.LightGBMClassifier
property), 1415
feature_importance(evalml.pipelines.components.LightGBMRegressor
property), 1418
feature_importance(evalml.pipelines.components.LinearRegressor
property), 1424
feature_importance(evalml.pipelines.components.LogisticRegressionClassifier
property), 1427
feature_importance(evalml.pipelines.components.ProphetRegressor
property), 1458
feature_importance(evalml.pipelines.components.RandomForestClassifier
property), 1461
feature_importance(evalml.pipelines.components.RandomForestRegressor
property), 1464
feature_importance(evalml.pipelines.components.StackedEnsembleBase
property), 1488
feature_importance(evalml.pipelines.components.StackedEnsembleClassifier
property), 1492
feature_importance(evalml.pipelines.components.StackedEnsembleRegressor
property), 1496
feature_importance(evalml.pipelines.components.SVMClassifier
property), 1507
feature_importance(evalml.pipelines.components.SVMRegressor
property), 1510
feature_importance(evalml.pipelines.components.TimeSeriesBaselineEstimator
property), 1518
feature_importance(evalml.pipelines.components.VowpalWabbitBinaryClassifier
property), 1539
feature_importance(evalml.pipelines.components.VowpalWabbitMulticlassClassifier
property), 1542
feature_importance(evalml.pipelines.components.VowpalWabbitRegressor
property), 1545
feature_importance(evalml.pipelines.components.XGBoostClassifier
property), 1548
feature_importance(evalml.pipelines.components.XGBoostRegressor
property), 1551
feature_importance(evalml.pipelines.DecisionTreeClassifier
property), 1673
feature_importance(evalml.pipelines.DecisionTreeRegressor
property), 1676
feature_importance(evalml.pipelines.ElasticNetClassifier
property), 1685
feature_importance(evalml.pipelines.ElasticNetRegressor
property), 1688
feature_importance(evalml.pipelines.Estimator
property), 1691
feature_importance(evalml.pipelines.ExponentialSmoothingRegressor
property), 1695
feature_importance(evalml.pipelines.ExtraTreesClassifier
property), 1699
feature_importance(evalml.pipelines.ExtraTreesRegressor
property), 1702
feature_importance(evalml.pipelines.KNeighborsClassifier
property), 1711
feature_importance(evalml.pipelines.LightGBMClassifier
property), 1715
feature_importance(evalml.pipelines.LightGBMRegressor
property), 1718
feature_importance(evalml.pipelines.LinearRegressor
property), 1721
feature_importance(evalml.pipelines.LogisticRegressionClassifier
property), 1725
feature_importance(evalml.pipelines.multiclass_classification_pipeline
property), 1577
feature_importance(evalml.pipelines.MulticlassClassificationPipeline
property), 1729
feature_importance(evalml.pipelines.pipeline_base.PipelineBase
property), 1584
feature_importance(evalml.pipelines.PipelineBase
property), 1744
feature_importance(evalml.pipelines.ProphetRegressor
property), 1749
feature_importance(evalml.pipelines.RandomForestClassifier
property), 1752
feature_importance(evalml.pipelines.RandomForestRegressor
property), 1755
feature_importance(evalml.pipelines.regression_pipeline.RegressionPi
property), 1592
feature_importance(evalml.pipelines.RegressionPipeline
property), 1760
feature_importance(evalml.pipelines.StackedEnsembleBase
property), 1773
feature_importance(evalml.pipelines.StackedEnsembleClassifier
property), 1777
feature_importance(evalml.pipelines.StackedEnsembleRegressor
property), 1781
feature_importance(evalml.pipelines.SVMClassifier
property), 1786
feature_importance(evalml.pipelines.SVMRegressor
property), 1790
feature_importance(evalml.pipelines.time_series_classification_pipelin

*property*), 1599
feature_importance (*evalml.pipelines.time_series_classification_pipeline.TimeSeriesClassificationPipeline property*), 1607
feature_importance (*evalml.pipelines.time_series_classification_pipeline.TimeSeriesMulticlassClassificationPipeline property*), 1614
feature_importance (*evalml.pipelines.time_series_pipeline.TimeSeriesPipelineBase property*), 1622
feature_importance (*evalml.pipelines.time_series_regression_pipeline.TimeSeriesRegressionPipeline property*), 1630
feature_importance (*evalml.pipelines.TimeSeriesBinaryClassificationPipeline property*), 1797
feature_importance (*evalml.pipelines.TimeSeriesClassificationPipeline property*), 1804
feature_importance (*evalml.pipelines.TimeSeriesMulticlassClassificationPipeline property*), 1818
feature_importance (*evalml.pipelines.TimeSeriesRegressionPipeline property*), 1825
feature_importance (*evalml.pipelines.VowpalWabbitBinaryClassifier property*), 1838
feature_importance (*evalml.pipelines.VowpalWabbitMulticlassClassifier property*), 1841
feature_importance (*evalml.pipelines.VowpalWabbitRegressor property*), 1844
feature_importance (*evalml.pipelines.XGBoostClassifier property*), 1848
feature_importance (*evalml.pipelines.XGBoostRegressor property*), 1851
FeatureSelector (*class in evalml.pipelines*), 1704
FeatureSelector (*class in evalml.pipelines.components*), 1401
FeatureSelector (*class in evalml.pipelines.components.transformers*), 1233
FeatureSelector (*class in evalml.pipelines.components.transformers.feature_selection*), 1017
FeatureSelector (*class in evalml.pipelines.components.transformers.feature_selection.feature_selector*), 997
find_confusion_matrix_per_thresholds() (*in module evalml.model_understanding*), 431
find_confusion_matrix_per_thresholds() (*in module evalml.model_understanding.decision_boundary*), 409
fit() (*evalml.pipelines.ARIMARegressor method*), 1644
fit() (*evalml.pipelines.binary_classification_pipeline.BinaryClassificationPipeline method*), 1556
fit() (*evalml.pipelines.BinaryClassificationPipeline method*), 1648
fit() (*evalml.pipelines.CatBoostClassifier method*), 1654
fit() (*evalml.pipelines.CatBoostRegressor method*), 1657
fit() (*evalml.pipelines.classification_pipeline.ClassificationPipeline method*), 1564
fit() (*evalml.pipelines.ClassificationPipeline method*), 1661
fit() (*evalml.pipelines.multiclass_classification_pipeline.MulticlassClassificationPipeline method*), 1571
fit() (*evalml.pipelines.pipeline_base.PipelineBase method*), 1668
fit() (*evalml.pipelines.components.ARIMARegressor method*), 1339
fit() (*evalml.pipelines.components.BaselineClassifier method*), 1342
fit() (*evalml.pipelines.components.BaselineRegressor method*), 1345
fit() (*evalml.pipelines.components.CatBoostClassifier method*), 1348
fit() (*evalml.pipelines.components.CatBoostRegressor method*), 1352
fit() (*evalml.pipelines.components.component_base.ComponentBase method*), 1325
fit() (*evalml.pipelines.components.ComponentBase method*), 1354
fit() (*evalml.pipelines.components.DateTimeFeaturizer method*), 1358
fit() (*evalml.pipelines.components.DecisionTreeClassifier method*), 1361
fit() (*evalml.pipelines.components.DecisionTreeRegressor method*), 1365
fit() (*evalml.pipelines.components.DFSTransformer method*), 1368
fit() (*evalml.pipelines.components.DropColumns method*), 1370
fit() (*evalml.pipelines.components.DropNaNRowsTransformer method*), 1373
fit() (*evalml.pipelines.components.DropNullColumns method*), 1375
fit() (*evalml.pipelines.components.DropRowsTransformer method*), 1378
fit() (*evalml.pipelines.components.ElasticNetClassifier method*), 1381
fit() (*evalml.pipelines.components.ElasticNetRegressor method*), 1384
fit() (*evalml.pipelines.components.EmailFeaturizer method*), 1386
fit() (*evalml.pipelines.components.ensemble.stacked_ensemble_base.StackedEnsembleBase method*), 621
fit() (*evalml.pipelines.components.ensemble.stacked_ensemble_classifier.StackedEnsembleClassifier method*), 625
fit() (*evalml.pipelines.components.ensemble.stacked_ensemble_regressor.StackedEnsembleRegressor method*), 629
fit() (*evalml.pipelines.components.ensemble.StackedEnsembleBase method*), 632
fit() (*evalml.pipelines.components.ensemble.StackedEnsembleClassifier method*), 636
fit() (*evalml.pipelines.components.ensemble.StackedEnsembleRegressor method*), 640

fit() (*evalml.pipelines.components.Estimator method*), 1389
fit() (*evalml.pipelines.components.estimators.ARIMARegressor method*), 861
fit() (*evalml.pipelines.components.estimators.BaselineClassifier method*), 864
fit() (*evalml.pipelines.components.estimators.BaselineRegressor method*), 867
fit() (*evalml.pipelines.components.estimators.CatBoostClassifier method*), 870
fit() (*evalml.pipelines.components.estimators.CatBoostRegressor method*), 874
fit() (*evalml.pipelines.components.estimators.classifiers.baseline_classifier.BaselineClassifier method*), 645
fit() (*evalml.pipelines.components.estimators.classifiers.BaselineClassifier method*), 698
fit() (*evalml.pipelines.components.estimators.classifiers.catboost_classifier.CatBoostClassifier method*), 648
fit() (*evalml.pipelines.components.estimators.classifiers.CatBoostClassifier method*), 701
fit() (*evalml.pipelines.components.estimators.classifiers.decision_tree_classifier.DecisionTreeClassifier method*), 652
fit() (*evalml.pipelines.components.estimators.classifiers.DecisionTreeClassifier method*), 704
fit() (*evalml.pipelines.components.estimators.classifiers.elastic_net_classifier.ElasticNetClassifier method*), 657
fit() (*evalml.pipelines.components.estimators.classifiers.ElasticNetClassifier method*), 708
fit() (*evalml.pipelines.components.estimators.classifiers.extra_trees_classifier.ExtraTreesClassifier method*), 661
fit() (*evalml.pipelines.components.estimators.classifiers.ExtraTreesClassifier method*), 712
fit() (*evalml.pipelines.components.estimators.classifiers.k_neighbors_classifier.KNeighborsClassifier method*), 665
fit() (*evalml.pipelines.components.estimators.classifiers.KNeighborsClassifier method*), 715
fit() (*evalml.pipelines.components.estimators.classifiers.lightgbm_classifier.LightGBMClassifier method*), 669
fit() (*evalml.pipelines.components.estimators.classifiers.LightGBMClassifier method*), 719
fit() (*evalml.pipelines.components.estimators.classifiers.logistic_regression_classifier.LogisticRegressionClassifier method*), 673
fit() (*evalml.pipelines.components.estimators.classifiers.LogisticRegressionClassifier method*), 722
fit() (*evalml.pipelines.components.estimators.classifiers.RandomForestClassifier method*), 726
fit() (*evalml.pipelines.components.estimators.classifiers.random_forest_classifier.RandomForestClassifier method*), 676
fit() (*evalml.pipelines.components.estimators.classifiers.svm_classifier.SVMClassifier method*), 680
fit() (*evalml.pipelines.components.estimators.classifiers.SVMClassifier method*), 729
fit() (*evalml.pipelines.components.estimators.classifiers.vowpal_wabbit_classifier.VowpalWabbitBaseClassifier method*), 684
fit() (*evalml.pipelines.components.estimators.classifiers.vowpal_wabbit_c method*), 687
fit() (*evalml.pipelines.components.estimators.classifiers.vowpal_wabbit_c method*), 690
fit() (*evalml.pipelines.components.estimators.classifiers.VowpalWabbitBi method*), 732
fit() (*evalml.pipelines.components.estimators.classifiers.VowpalWabbitM method*), 736
fit() (*evalml.pipelines.components.estimators.classifiers.xgboost_classifie method*), 694
fit() (*evalml.pipelines.components.estimators.classifiers.XGBoostClassifi method*), 739
fit() (*evalml.pipelines.components.estimators.DecisionTreeClassifier method*), 877
fit() (*evalml.pipelines.components.estimators.DecisionTreeRegressor method*), 881
fit() (*evalml.pipelines.components.estimators.ElasticNetClassifier method*), 885
fit() (*evalml.pipelines.components.estimators.ElasticNetRegressor method*), 888
fit() (*evalml.pipelines.components.estimators.Estimator method*), 891
fit() (*evalml.pipelines.components.estimator.Estimator method*), 855
fit() (*evalml.pipelines.components.estimators.ExponentialSmoothingRegr method*), 895
fit() (*evalml.pipelines.components.estimators.ExtraTreesClassifier method*), 898
fit() (*evalml.pipelines.components.estimators.ExtraTreesRegressor method*), 902
fit() (*evalml.pipelines.components.estimators.KNeighborsClassifier method*), 906
fit() (*evalml.pipelines.components.estimators.LightGBMClassifier method*), 910
fit() (*evalml.pipelines.components.estimators.LightGBMRegressor method*), 913
fit() (*evalml.pipelines.components.estimators.LinearRegressor method*), 916
fit() (*evalml.pipelines.components.estimators.LogisticRegressionClassifie method*), 920
fit() (*evalml.pipelines.components.estimators.LogisticRegressionClassifier Regressor method*), 923
fit() (*evalml.pipelines.components.estimators.RandomForestClassifier method*), 926
fit() (*evalml.pipelines.components.estimators.RandomForestRegressor method*), 930
fit() (*evalml.pipelines.components.estimators.regressors.arima_regressor method*), 744
fit() (*evalml.pipelines.components.estimators.regressors.ARIMARegresso method*), 804
fit() (*evalml.pipelines.components.estimators.regressors.baseline_regress method*), 748
fit() (*evalml.pipelines.components.estimators.regressors.BaselineRegress method*), 807

fit() (evalml.pipelines.components.estimators.regressors.catboost_regressor.CatBoostRegressor method), 752
fit() (evalml.pipelines.components.estimators.regressors.CatBoostRegressor method), 810
fit() (evalml.pipelines.components.estimators.regressors.decision_tree_regressor.DecisionTreeRegressor method), 756
fit() (evalml.pipelines.components.estimators.regressors.DecisionTreeRegressor method), 814
fit() (evalml.pipelines.components.estimators.regressors.elastic_net_regressor.ElasticNetRegressor method), 759
fit() (evalml.pipelines.components.estimators.regressors.ElasticNetRegressor method), 817
fit() (evalml.pipelines.components.estimators.regressors.extra_trees_regressor.ExtraTreesRegressor method), 764
fit() (evalml.pipelines.components.estimators.regressors.ExtraTreesRegressor method), 821
fit() (evalml.pipelines.components.estimators.regressors.exponential_smoothing_regressor.ExponentialSmoothingRegressor method), 768
fit() (evalml.pipelines.components.estimators.regressors.ExponentialSmoothingRegressor method), 825
fit() (evalml.pipelines.components.estimators.regressors.lightgbm_regressor.LightGBMRegressor method), 772
fit() (evalml.pipelines.components.estimators.regressors.LightGBMRegressor method), 828
fit() (evalml.pipelines.components.estimators.regressors.linear_regressor.LinearRegressor method), 775
fit() (evalml.pipelines.components.estimators.regressors.LinearRegressor method), 831
fit() (evalml.pipelines.components.estimators.regressors.prophet_regressor.ProphetRegressor method), 780
fit() (evalml.pipelines.components.estimators.regressors.ProphetRegressor method), 835
fit() (evalml.pipelines.components.estimators.regressors.random_forest_regressor.RandomForestRegressor method), 839
fit() (evalml.pipelines.components.estimators.regressors.rf_regressor.RandomForestRegressor method), 784
fit() (evalml.pipelines.components.estimators.regressors.svm_regressor.SVMRegressor method), 788
fit() (evalml.pipelines.components.estimators.regressors.SVMRegressor method), 842
fit() (evalml.pipelines.components.estimators.regressors.time_series_baseline_estimator.TimeSeriesBaselineEstimator method), 791
fit() (evalml.pipelines.components.estimators.regressors.TimeSeriesBaselineEstimator method), 845
fit() (evalml.pipelines.components.estimators.regressors.vowpal_wabbit_regressor.VowpalWabbitRegressor method), 795
fit() (evalml.pipelines.components.estimators.regressors.VowpalWabbitRegressor method), 848
fit() (evalml.pipelines.components.estimators.regressors.xgboost_regressor.XGBoostRegressor method), 799
fit() (evalml.pipelines.components.estimators.regressors.XGBoostRegressor method), 852
fit() (evalml.pipelines.components.estimators.SVMClassifier method), 933
fit() (evalml.pipelines.components.estimators.SVMRegressor method), 936
fit() (evalml.pipelines.components.estimators.TimeSeriesBaselineEstimator method), 939
fit() (evalml.pipelines.components.estimators.VowpalWabbitBinaryClassifier method), 942
fit() (evalml.pipelines.components.estimators.VowpalWabbitMulticlassClassifier method), 946
fit() (evalml.pipelines.components.estimators.VowpalWabbitRegressor method), 949
fit() (evalml.pipelines.components.estimators.XGBoostClassifier method), 952
fit() (evalml.pipelines.components.estimators.XGBoostRegressor method), 955
fit() (evalml.pipelines.components.ExponentialSmoothingRegressor method), 1393
fit() (evalml.pipelines.components.ExtraTreesClassifier method), 1396
fit() (evalml.pipelines.components.ExtraTreesRegressor method), 1400
fit() (evalml.pipelines.components.FeatureSelector method), 1403
fit() (evalml.pipelines.components.Imputer method), 1406
fit() (evalml.pipelines.components.KNeighborsClassifier method), 1409
fit() (evalml.pipelines.components.LabelEncoder method), 1412
fit() (evalml.pipelines.components.LightGBMClassifier method), 1415
fit() (evalml.pipelines.components.LightGBMRegressor method), 1418
fit() (evalml.pipelines.components.LinearDiscriminantAnalysis method), 1421
fit() (evalml.pipelines.components.LinearRegressor method), 1424
fit() (evalml.pipelines.components.LogisticRegressionClassifier method), 1427
fit() (evalml.pipelines.components.LogTransformer method), 1430
fit() (evalml.pipelines.components.TimeSeriesBaselineEstimator method), 1433
fit() (evalml.pipelines.components.NaturalLanguageFeaturizer method), 1435
fit() (evalml.pipelines.components.OneHotEncoder method), 1438
fit() (evalml.pipelines.components.OrdinalEncoder method), 1442
fit() (evalml.pipelines.components.Oversampler method), 1445
fit() (evalml.pipelines.components.PCA method), 1447
fit() (evalml.pipelines.components.PerColumnImputer method), 1449
fit() (evalml.pipelines.components.PolynomialDecomposer method), 1453

fit() (evalml.pipelines.components.ProphetRegressor method), 1458
fit() (evalml.pipelines.components.RandomForestClassifier method), 1461
fit() (evalml.pipelines.components.RandomForestRegressor method), 1464
fit() (evalml.pipelines.components.ReplaceNullableTypes method), 1467
fit() (evalml.pipelines.components.RFClassifierRFESelector method), 1469
fit() (evalml.pipelines.components.RFClassifierSelectFromModel method), 1472
fit() (evalml.pipelines.components.RFRegressorRFESelector method), 1475
fit() (evalml.pipelines.components.RFRegressorSelectFromModel method), 1478
fit() (evalml.pipelines.components.SelectByType method), 1481
fit() (evalml.pipelines.components.SelectColumns method), 1483
fit() (evalml.pipelines.components.SimpleImputer method), 1486
fit() (evalml.pipelines.components.StackedEnsembleBase method), 1488
fit() (evalml.pipelines.components.StackedEnsembleClassifier method), 1492
fit() (evalml.pipelines.components.StackedEnsembleRegressor method), 1496
fit() (evalml.pipelines.components.StandardScaler method), 1499
fit() (evalml.pipelines.components.STLDecomposer method), 1502
fit() (evalml.pipelines.components.SVMClassifier method), 1507
fit() (evalml.pipelines.components.SVMRegressor method), 1510
fit() (evalml.pipelines.components.TargetEncoder method), 1513
fit() (evalml.pipelines.components.TargetImputer method), 1516
fit() (evalml.pipelines.components.TimeSeriesBaselineEstimator method), 1518
fit() (evalml.pipelines.components.TimeSeriesFeaturizer method), 1522
fit() (evalml.pipelines.components.TimeSeriesImputer method), 1525
fit() (evalml.pipelines.components.TimeSeriesRegularizer method), 1528
fit() (evalml.pipelines.components.Transformer method), 1530
fit() (evalml.pipelines.components.transformers.column_selectors.ColumnSelector method), 1202
fit() (evalml.pipelines.components.transformers.column_selectors.DropColumns method), 1205
fit() (evalml.pipelines.components.transformers.column_selectors.SelectE method), 1207
fit() (evalml.pipelines.components.transformers.column_selectors.SelectC method), 1209
fit() (evalml.pipelines.components.transformers.DateTimeFeaturizer method), 1217
fit() (evalml.pipelines.components.transformers.DFSTransformer method), 1220
fit() (evalml.pipelines.components.transformers.dimensionality_reduction method), 958
fit() (evalml.pipelines.components.transformers.dimensionality_reduction method), 964
fit() (evalml.pipelines.components.transformers.dimensionality_reduction method), 967
fit() (evalml.pipelines.components.transformers.dimensionality_reduction method), 961
fit() (evalml.pipelines.components.transformers.DropColumns method), 1222
fit() (evalml.pipelines.components.transformers.DropNaNRowsTransform method), 1224
fit() (evalml.pipelines.components.transformers.DropNullColumns method), 1227
fit() (evalml.pipelines.components.transformers.DropRowsTransformer method), 1229
fit() (evalml.pipelines.components.transformers.EmailFeaturizer method), 1232
fit() (evalml.pipelines.components.transformers.encoders.label_encoder.I method), 970
fit() (evalml.pipelines.components.transformers.encoders.LabelEncoder method), 985
fit() (evalml.pipelines.components.transformers.encoders.onehot_encode method), 973
fit() (evalml.pipelines.components.transformers.encoders.OneHotEncode method), 989
fit() (evalml.pipelines.components.transformers.encoders.ordinal_encode method), 978
fit() (evalml.pipelines.components.transformers.encoders.OrdinalEncode method), 992
fit() (evalml.pipelines.components.transformers.encoders.target_encoder. method), 982
fit() (evalml.pipelines.components.transformers.encoders.TargetEncoder method), 995
fit() (evalml.pipelines.components.transformers.feature_selection.feature_ method), 998
fit() (evalml.pipelines.components.transformers.feature_selection.Feature method), 1018
fit() (evalml.pipelines.components.transformers.feature_selection.recursi method), 1001
fit() (evalml.pipelines.components.transformers.feature_selection.recursi method), 1005
fit() (evalml.pipelines.components.transformers.feature_selection.recursi method), 1008
fit() (evalml.pipelines.components.transformers.feature_selection.rf_clas method), 1011

fit() (evalml.pipelines.components.transformers.feature_selection.RFClassifierSelectFromModel method), 1015
fit() (evalml.pipelines.components.transformers.feature_selection.RFClassifierSelector method), 1021
fit() (evalml.pipelines.components.transformers.feature_selection.RFRegressorSelectFromModel method), 1024
fit() (evalml.pipelines.components.transformers.feature_selection.RFRegressorSelector method), 1028
fit() (evalml.pipelines.components.transformers.feature_selection.RandomForestClassifierSelectFromModel method), 1031
fit() (evalml.pipelines.components.transformers.FeatureSelector method), 1234
fit() (evalml.pipelines.components.transformers.Imputer method), 1237
fit() (evalml.pipelines.components.transformers.imputers.Imputer method), 1053
fit() (evalml.pipelines.components.transformers.imputers.TimeSeriesImputer method), 1034
fit() (evalml.pipelines.components.transformers.imputers.KNNImputer method), 1037
fit() (evalml.pipelines.components.transformers.imputers.KNNImputer method), 1056
fit() (evalml.pipelines.components.transformers.imputers.PerColumnImputer method), 1040
fit() (evalml.pipelines.components.transformers.imputers.PerColumnImputer method), 1058
fit() (evalml.pipelines.components.transformers.imputers.SimpleImputer method), 1043
fit() (evalml.pipelines.components.transformers.imputers.SimpleImputer method), 1061
fit() (evalml.pipelines.components.transformers.imputers.TargetImputer method), 1046
fit() (evalml.pipelines.components.transformers.imputers.TargetImputer method), 1064
fit() (evalml.pipelines.components.transformers.imputers.TimeSeriesImputer method), 1050
fit() (evalml.pipelines.components.transformers.imputers.TimeSeriesImputer method), 1066
fit() (evalml.pipelines.components.transformers.LabelEncoder method), 1240
fit() (evalml.pipelines.components.transformers.LinearDiscriminantAnalysis method), 1242
fit() (evalml.pipelines.components.transformers.LogTransformer method), 1245
fit() (evalml.pipelines.components.transformers.LSA method), 1247
fit() (evalml.pipelines.components.transformers.NaturalLanguageFeaturizer method), 1250
fit() (evalml.pipelines.components.transformers.OneHotEncoder method), 1253
fit() (evalml.pipelines.components.transformers.OrdinalEncoder method), 1256
fit() (evalml.pipelines.components.transformers.Oversampler method), 1259
fit() (evalml.pipelines.components.transformers.PCA_Decomposer method), 1262
fit() (evalml.pipelines.components.transformers.PerColumnImputer method), 1264
fit() (evalml.pipelines.components.transformers.PolynomialDecomposer method), 1268
fit() (evalml.pipelines.components.transformers.preprocessing.datetime_f method), 1070
fit() (evalml.pipelines.components.transformers.preprocessing.DateTimeF method), 1130
fit() (evalml.pipelines.components.transformers.preprocessing.Decompos method), 1133
fit() (evalml.pipelines.components.transformers.preprocessing.decompose method), 1074
fit() (evalml.pipelines.components.transformers.preprocessing.DFSTransf method), 1137
fit() (evalml.pipelines.components.transformers.preprocessing.drop_nan_ method), 1077
fit() (evalml.pipelines.components.transformers.preprocessing.drop_null_ method), 1080
fit() (evalml.pipelines.components.transformers.preprocessing.drop_rows method), 1083
fit() (evalml.pipelines.components.transformers.preprocessing.DropNaNR method), 1139
fit() (evalml.pipelines.components.transformers.preprocessing.DropNullC method), 1142
fit() (evalml.pipelines.components.transformers.preprocessing.DropRows method), 1144
fit() (evalml.pipelines.components.transformers.preprocessing.EmailFeat method), 1147
fit() (evalml.pipelines.components.transformers.preprocessing.featuretool method), 1087
fit() (evalml.pipelines.components.transformers.preprocessing.log_transf method), 1089
fit() (evalml.pipelines.components.transformers.preprocessing.LogTransf method), 1149
fit() (evalml.pipelines.components.transformers.preprocessing.LSA method), 1151
fit() (evalml.pipelines.components.transformers.preprocessing.lsa.LSA method), 1092
fit() (evalml.pipelines.components.transformers.preprocessing.natural_la method), 1095
fit() (evalml.pipelines.components.transformers.preprocessing.NaturalLa method), 1154
fit() (evalml.pipelines.components.transformers.preprocessing.polynomia method), 1100
fit() (evalml.pipelines.components.transformers.preprocessing.Polynomia method), 1157
fit() (evalml.pipelines.components.transformers.preprocessing.replace_nu method), 1104
fit() (evalml.pipelines.components.transformers.preprocessing.ReplaceNu method), 1162
fit() (evalml.pipelines.components.transformers.preprocessing.stl_decomp method), 1109

fit() (*evalml.pipelines.components.transformers.preprocessing.STLDecomposer method*), 1165
fit() (*evalml.pipelines.components.transformers.preprocessing.TextTransformer method*), 1113
fit() (*evalml.pipelines.components.transformers.preprocessing.TimeSeriesFeaturizer method*), 1169
fit() (*evalml.pipelines.components.transformers.preprocessing.TimeSeriesImputer method*), 1117
fit() (*evalml.pipelines.components.transformers.preprocessing.TimeSeriesRegularizer method*), 1121
fit() (*evalml.pipelines.components.transformers.preprocessing.TimeSeriesRemoverer method*), 1173
fit() (*evalml.pipelines.components.transformers.preprocessing.TimeSeriesSplitter method*), 1176
fit() (*evalml.pipelines.components.transformers.preprocessing.transform_primitive_components.EmailFeaturizer method*), 1124
fit() (*evalml.pipelines.components.transformers.preprocessing.transform_primitive_components.URLFeaturizer method*), 1126
fit() (*evalml.pipelines.components.transformers.preprocessing.URLFeaturizer method*), 1178
fit() (*evalml.pipelines.components.transformers.ReplaceNullsBeforeTypes method*), 1272
fit() (*evalml.pipelines.components.transformers.RFClassificationSelector method*), 1275
fit() (*evalml.pipelines.components.transformers.RFClassificationSelectFromModel method*), 1278
fit() (*evalml.pipelines.components.transformers.RFRegressionSelector method*), 1281
fit() (*evalml.pipelines.components.transformers.RFRegressionSelectFromModel method*), 1284
fit() (*evalml.pipelines.components.transformers.samplers.base_sampler.BaseSampler method*), 1181
fit() (*evalml.pipelines.components.transformers.samplers.OversampleSampler method*), 1191
fit() (*evalml.pipelines.components.transformers.samplers.OversamplerBase method*), 1184
fit() (*evalml.pipelines.components.transformers.samplers.SMOTENCSampler method*), 1194
fit() (*evalml.pipelines.components.transformers.samplers.SMOTENSampler method*), 1188
fit() (*evalml.pipelines.components.transformers.scalers.StandardScaler method*), 1197
fit() (*evalml.pipelines.components.transformers.scalers.StandardScalerBase method*), 1199
fit() (*evalml.pipelines.components.transformers.SelectByType method*), 1286
fit() (*evalml.pipelines.components.transformers.SelectColumns method*), 1288
fit() (*evalml.pipelines.components.transformers.SimpleImputer method*), 1291
fit() (*evalml.pipelines.components.transformers.StandardScaler method*), 1293
fit() (*evalml.pipelines.components.transformers.STLDecomposer method*), 1297
fit() (*evalml.pipelines.components.transformers.TargetEncoder method*), 1302
fit() (*evalml.pipelines.components.transformers.TargetImputer method*), 1304
fit() (*evalml.pipelines.components.transformers.TimeSeriesFeaturizer method*), 1308
fit() (*evalml.pipelines.components.transformers.TimeSeriesImputer method*), 1310
fit() (*evalml.pipelines.components.transformers.TimeSeriesRegularizer method*), 1313
fit() (*evalml.pipelines.components.transformers.Transformer method*), 1316
fit() (*evalml.pipelines.components.transformers.transformer.Transformer method*), 1212
fit() (*evalml.pipelines.components.transformers.transform_primitive_components.EmailFeaturizer method*), 1320
fit() (*evalml.pipelines.components.transformers.transform_primitive_components.URLFeaturizer method*), 1322
fit() (*evalml.pipelines.components.Undersampler method*), 1533
fit() (*evalml.pipelines.components.URLFeaturizer method*), 1536
fit() (*evalml.pipelines.components.utils.WrappedSKClassifier method*), 1331
fit() (*evalml.pipelines.components.utils.WrappedSKRegressor method*), 1332
fit() (*evalml.pipelines.components.VowpalWabbitBinaryClassifier method*), 1539
fit() (*evalml.pipelines.components.VowpalWabbitMulticlassClassifier method*), 1542
fit() (*evalml.pipelines.components.VowpalWabbitRegressor method*), 1545
fit() (*evalml.pipelines.components.XGBoostClassifier method*), 1548
fit() (*evalml.pipelines.components.XGBoostRegressor method*), 1551
fit() (*evalml.pipelines.DecisionTreeClassifier method*), 1673
fit() (*evalml.pipelines.DecisionTreeRegressor method*), 1676
fit() (*evalml.pipelines.DelayedFeatureTransformer method*), 1680
fit() (*evalml.pipelines.DropNaNRowsTransformer method*), 1682
fit() (*evalml.pipelines.ElasticNetClassifier method*), 1685
fit() (*evalml.pipelines.ElasticNetRegressor method*), 1688
fit() (*evalml.pipelines.Estimator method*), 1691
fit() (*evalml.pipelines.ExponentialSmoothingRegressor method*), 1695
fit() (*evalml.pipelines.ExtraTreesClassifier method*), 1699
fit() (*evalml.pipelines.ExtraTreesRegressor method*), 1702

fit() (*evalml.pipelines.FeatureSelector method*), 1705
fit() (*evalml.pipelines.Imputer method*), 1708
fit() (*evalml.pipelines.KNeighborsClassifier method*), 1711
fit() (*evalml.pipelines.LightGBMClassifier method*), 1715
fit() (*evalml.pipelines.LightGBMRegressor method*), 1718
fit() (*evalml.pipelines.LinearRegressor method*), 1721
fit() (*evalml.pipelines.LogisticRegressionClassifier method*), 1725
fit() (*evalml.pipelines.multiclass_classification_pipeline.MulticlassClassificationPipeline method*), 1577
fit() (*evalml.pipelines.MulticlassClassificationPipeline method*), 1729
fit() (*evalml.pipelines.OneHotEncoder method*), 1735
fit() (*evalml.pipelines.OrdinalEncoder method*), 1738
fit() (*evalml.pipelines.PerColumnImputer method*), 1741
fit() (*evalml.pipelines.pipeline_base.PipelineBase method*), 1584
fit() (*evalml.pipelines.PipelineBase method*), 1744
fit() (*evalml.pipelines.ProphetRegressor method*), 1749
fit() (*evalml.pipelines.RandomForestClassifier method*), 1752
fit() (*evalml.pipelines.RandomForestRegressor method*), 1755
fit() (*evalml.pipelines.regression_pipeline.RegressionPipeline method*), 1592
fit() (*evalml.pipelines.RegressionPipeline method*), 1760
fit() (*evalml.pipelines.RFClassifierSelectFromModel method*), 1765
fit() (*evalml.pipelines.RFRegressorSelectFromModel method*), 1768
fit() (*evalml.pipelines.SimpleImputer method*), 1771
fit() (*evalml.pipelines.StackedEnsembleBase method*), 1773
fit() (*evalml.pipelines.StackedEnsembleClassifier method*), 1777
fit() (*evalml.pipelines.StackedEnsembleRegressor method*), 1781
fit() (*evalml.pipelines.StandardScaler method*), 1784
fit() (*evalml.pipelines.SVMClassifier method*), 1786
fit() (*evalml.pipelines.SVMRegressor method*), 1790
fit() (*evalml.pipelines.TargetEncoder method*), 1793
fit() (*evalml.pipelines.time_series_classification_pipelines.TimeSeriesBinaryClassificationPipeline method*), 1599
fit() (*evalml.pipelines.time_series_classification_pipelines.TimeSeriesClassificationPipeline method*), 1607
fit() (*evalml.pipelines.time_series_classification_pipelines.TimeSeriesMulticlassClassificationPipeline method*), 1614
fit() (*evalml.pipelines.time_series_pipeline_base.TimeSeriesPipelineBase method*), 1622
fit() (*evalml.pipelines.time_series_regression_pipeline.TimeSeriesRegressionPipeline method*), 1630
fit() (*evalml.pipelines.TimeSeriesBinaryClassificationPipeline method*), 1797
fit() (*evalml.pipelines.TimeSeriesClassificationPipeline method*), 1805
fit() (*evalml.pipelines.TimeSeriesFeaturizer method*), 1811
fit() (*evalml.pipelines.TimeSeriesImputer method*), 1814
fit() (*evalml.pipelines.TimeSeriesMulticlassClassificationPipeline method*), 1818
fit() (*evalml.pipelines.TimeSeriesRegressionPipeline method*), 1826
fit() (*evalml.pipelines.TimeSeriesRegularizer method*), 1833
fit() (*evalml.pipelines.Transformer method*), 1835
fit() (*evalml.pipelines.VowpalWabbitBinaryClassifier method*), 1838
fit() (*evalml.pipelines.VowpalWabbitMulticlassClassifier method*), 1841
fit() (*evalml.pipelines.VowpalWabbitRegressor method*), 1844
fit() (*evalml.pipelines.XGBoostClassifier method*), 1848
fit() (*evalml.pipelines.XGBoostRegressor method*), 1851
fit_and_transform_all_but_final() (*evalml.pipelines.component_graph.ComponentGraph method*), 1571
fit_and_transform_all_but_final() (*evalml.pipelines.ComponentGraph method*), 1668
fit_resample() (*evalml.pipelines.components.transformers.samplers.Undersampler method*), 1194
fit_resample() (*evalml.pipelines.components.transformers.samplers.Undersampler method*), 1188
fit_resample() (*evalml.pipelines.components.transformers.Undersampler method*), 1320
fit_resample() (*evalml.pipelines.components.Undersampler method*), 1533
fit_transform() (*evalml.pipelines.binary_classification_pipeline.BinaryClassificationPipeline method*), 1556
fit_transform() (*evalml.pipelines.BinaryClassificationPipeline method*), 1648
fit_transform() (*evalml.pipelines.classification_pipeline.ClassificationPipeline method*), 1564
fit_transform() (*evalml.pipelines.ClassificationPipeline method*), 1661
fit_transform() (*evalml.pipelines.component_graph.ComponentGraph method*), 1573
fit_transform() (*evalml.pipelines.ComponentGraph method*), 1668
fit_transform() (*evalml.pipelines.components.DateTimeFeaturizer*

*method*), 1358
fit_transform() (*evalml.pipelines.components.DFSTransformer method*), 1368
fit_transform() (*evalml.pipelines.components.DropColumns method*), 1371
fit_transform() (*evalml.pipelines.components.DropNaNRowsTransformer method*), 1373
fit_transform() (*evalml.pipelines.components.DropNullColumns method*), 1375
fit_transform() (*evalml.pipelines.components.DropRowsTransformer method*), 1378
fit_transform() (*evalml.pipelines.components.EmailFeaturizer method*), 1386
fit_transform() (*evalml.pipelines.components.FeatureSelector method*), 1403
fit_transform() (*evalml.pipelines.components.Imputer method*), 1406
fit_transform() (*evalml.pipelines.components.LabelEncoder method*), 1412
fit_transform() (*evalml.pipelines.components.LinearDiscriminantAnalysis method*), 1421
fit_transform() (*evalml.pipelines.components.LogTransformer method*), 1430
fit_transform() (*evalml.pipelines.components.LSA method*), 1433
fit_transform() (*evalml.pipelines.components.NaturalLanguageFeaturizer method*), 1435
fit_transform() (*evalml.pipelines.components.OneHotEncoder method*), 1439
fit_transform() (*evalml.pipelines.components.OrdinalEncoder method*), 1442
fit_transform() (*evalml.pipelines.components.Oversampler method*), 1445
fit_transform() (*evalml.pipelines.components.PCA method*), 1447
fit_transform() (*evalml.pipelines.components.PerColumnImputer method*), 1450
fit_transform() (*evalml.pipelines.components.PolynomialDecomposer method*), 1453
fit_transform() (*evalml.pipelines.components.ReplaceNullableTypes method*), 1467
fit_transform() (*evalml.pipelines.components.RFClassifierRFESelector method*), 1470
fit_transform() (*evalml.pipelines.components.RFClassifierSelectFromModel method*), 1473
fit_transform() (*evalml.pipelines.components.RFRegressorRFESelector method*), 1476
fit_transform() (*evalml.pipelines.components.RFRegressorSelectFromModel method*), 1479
fit_transform() (*evalml.pipelines.components.SelectByType method*), 1481
fit_transform() (*evalml.pipelines.components.SelectColumns method*), 1483
fit_transform() (*evalml.pipelines.components.SimpleImputer method*), 1486
fit_transform() (*evalml.pipelines.components.StandardScaler method*), 1499
fit_transform() (*evalml.pipelines.components.STLDecomposer method*), 1502
fit_transform() (*evalml.pipelines.components.TargetEncoder method*), 1513
fit_transform() (*evalml.pipelines.components.TargetImputer method*), 1516
fit_transform() (*evalml.pipelines.components.TimeSeriesFeaturizer method*), 1522
fit_transform() (*evalml.pipelines.components.TimeSeriesImputer method*), 1525
fit_transform() (*evalml.pipelines.components.TimeSeriesRegularizer method*), 1528
fit_transform() (*evalml.pipelines.components.Transformer method*), 1530
fit_transform() (*evalml.pipelines.components.transformers.column_sele method*), 1202
fit_transform() (*evalml.pipelines.components.transformers.column_sele method*), 1205
fit_transform() (*evalml.pipelines.components.transformers.column_sele method*), 1207
fit_transform() (*evalml.pipelines.components.transformers.column_sele method*), 1210
fit_transform() (*evalml.pipelines.components.transformers.DateTimeFe method*), 1217
fit_transform() (*evalml.pipelines.components.transformers.DFSTransfo method*), 1220
fit_transform() (*evalml.pipelines.components.transformers.dimensiona method*), 958
fit_transform() (*evalml.pipelines.components.transformers.dimensiona method*), 964
fit_transform() (*evalml.pipelines.components.transformers.dimensiona method*), 967
fit_transform() (*evalml.pipelines.components.transformers.dimensiona method*), 961
fit_transform() (*evalml.pipelines.components.transformers.DropColum method*), 1222
fit_transform() (*evalml.pipelines.components.transformers.DropNaNRo method*), 1225
fit_transform() (*evalml.pipelines.components.transformers.DropNullCo method*), 1227
fit_transform() (*evalml.pipelines.components.transformers.DropRowsT method*), 1229
fit_transform() (*evalml.pipelines.components.transformers.EmailFeatu method*), 1232
fit_transform() (*evalml.pipelines.components.transformers.encoders.la method*), 970
fit_transform() (*evalml.pipelines.components.transformers.encoders.La method*), 985
fit_transform() (*evalml.pipelines.components.transformers.encoders.on method*), 974
fit_transform() (*evalml.pipelines.components.transformers.encoders.On*

*method*), 989
fit_transform() (*evalml.pipelines.components.transformers.*
*method*), 978
fit_transform() (*evalml.pipelines.components.transformers.*
*method*), 992
fit_transform() (*evalml.pipelines.components.transformers.*
*method*), 982
fit_transform() (*evalml.pipelines.components.transformers.*
*method*), 995
fit_transform() (*evalml.pipelines.components.transformers.*
*method*), 998
fit_transform() (*evalml.pipelines.components.transformers.*
*method*), 1018
fit_transform() (*evalml.pipelines.components.transformers.*
*method*), 1002
fit_transform() (*evalml.pipelines.components.transformers.*
*method*), 1005
fit_transform() (*evalml.pipelines.components.transformers.*
*method*), 1008
fit_transform() (*evalml.pipelines.components.transformers.*
*method*), 1012
fit_transform() (*evalml.pipelines.components.transformers.*
*method*), 1015
fit_transform() (*evalml.pipelines.components.transformers.*
*method*), 1022
fit_transform() (*evalml.pipelines.components.transformers.*
*method*), 1025
fit_transform() (*evalml.pipelines.components.transformers.*
*method*), 1028
fit_transform() (*evalml.pipelines.components.transformers.*
*method*), 1031
fit_transform() (*evalml.pipelines.components.transformers.*
*method*), 1235
fit_transform() (*evalml.pipelines.components.transformers.*
*method*), 1237
fit_transform() (*evalml.pipelines.components.transformers.*
*method*), 1054
fit_transform() (*evalml.pipelines.components.transformers.*
*method*), 1034
fit_transform() (*evalml.pipelines.components.transformers.*
*method*), 1037
fit_transform() (*evalml.pipelines.components.transformers.*
*method*), 1056
fit_transform() (*evalml.pipelines.components.transformers.*
*method*), 1040
fit_transform() (*evalml.pipelines.components.transformers.*
*method*), 1058
fit_transform() (*evalml.pipelines.components.transformers.*
*method*), 1043
fit_transform() (*evalml.pipelines.components.transformers.*
*method*), 1061
fit_transform() (*evalml.pipelines.components.transformers.*
*method*), 1047
fit_transform() (*evalml.pipelines.components.transformers.*
*method*), 1064
fit_transform() (*evalml.pipelines.components.transformers.imputers.ti*
*method*), 1051
fit_transform() (*evalml.pipelines.components.transformers.imputers.Ti*
*method*), 1067
fit_transform() (*evalml.pipelines.components.transformers.LabelEnco*
*method*), 1240
fit_transform() (*evalml.pipelines.components.transformers.LinearDisc*
*method*), 1242
fit_transform() (*evalml.pipelines.components.transformers.LogTransfo*
*method*), 1245
fit_transform() (*evalml.pipelines.components.transformers.LSA*
*method*), 1247
fit_transform() (*evalml.pipelines.components.transformers.*
*method*), 1250
fit_transform() (*evalml.pipelines.components.transformers.*
*method*), 1253
fit_transform() (*evalml.pipelines.components.transformers.*
*method*), 1256
fit_transform() (*evalml.pipelines.components.transformers.*
*method*), 1259
fit_transform() (*evalml.pipelines.components.transformers.*
*method*), 1262
fit_transform() (*evalml.pipelines.components.transformers.PerColumn*
*method*), 1264
fit_transform() (*evalml.pipelines.components.transformers.Polynomial*
*method*), 1268
fit_transform() (*evalml.pipelines.components.transformers.preprocessi*
*method*), 1070
fit_transform() (*evalml.pipelines.components.transformers.preprocessi*
*method*), 1130
fit_transform() (*evalml.pipelines.components.transformers.preprocessi*
*method*), 1133
fit_transform() (*evalml.pipelines.components.transformers.preprocessi*
*method*), 1074
fit_transform() (*evalml.pipelines.components.transformers.preprocessi*
*method*), 1137
fit_transform() (*evalml.pipelines.components.transformers.preprocessi*
*method*), 1078
fit_transform() (*evalml.pipelines.components.transformers.preprocessi*
*method*), 1080
fit_transform() (*evalml.pipelines.components.transformers.preprocessi*
*method*), 1083
fit_transform() (*evalml.pipelines.components.transformers.preprocessi*
*method*), 1139
fit_transform() (*evalml.pipelines.components.transformers.preprocessi*
*method*), 1142
fit_transform() (*evalml.pipelines.components.transformers.preprocessi*
*method*), 1144
fit_transform() (*evalml.pipelines.components.transformers.preprocessi*
*method*), 1147
fit_transform() (*evalml.pipelines.components.transformers.preprocessi*
*method*), 1087
fit_transform() (*evalml.pipelines.components.transformers.preprocessi*

fit_transform() (evalml.pipelines.components.transformers.LogTransformer method), 1089
fit_transform() (evalml.pipelines.components.transformers.LSA method), 1149
fit_transform() (evalml.pipelines.components.transformers.LSA method), 1151
fit_transform() (evalml.pipelines.components.transformers.LSA method), 1092
fit_transform() (evalml.pipelines.components.transformers.NaturalLanguageFeaturizer method), 1095
fit_transform() (evalml.pipelines.components.transformers.NaturalLanguageFeaturizer method), 1154
fit_transform() (evalml.pipelines.components.transformers.PolynomialDecomposer method), 1100
fit_transform() (evalml.pipelines.components.transformers.PolynomialDecomposer method), 1158
fit_transform() (evalml.pipelines.components.transformers.RemoveNullRowsTransformer method), 1104
fit_transform() (evalml.pipelines.components.transformers.RemoveNullRowsTransformer method), 1162
fit_transform() (evalml.pipelines.components.transformers.STLDecomposer method), 1109
fit_transform() (evalml.pipelines.components.transformers.STLDecomposer method), 1165
fit_transform() (evalml.pipelines.components.transformers.TextFeaturizer method), 1114
fit_transform() (evalml.pipelines.components.transformers.TextFeaturizer method), 1169
fit_transform() (evalml.pipelines.components.transformers.TimeSeriesFeaturizer method), 1117
fit_transform() (evalml.pipelines.components.transformers.TimeSeriesFeaturizer method), 1121
fit_transform() (evalml.pipelines.components.transformers.TimeSeriesFeaturizer method), 1173
fit_transform() (evalml.pipelines.components.transformers.TimeSeriesRegularizer method), 1176
fit_transform() (evalml.pipelines.components.transformers.transformer_base.EmptyFeatureUnderSampler method), 1124
fit_transform() (evalml.pipelines.components.transformers.transformer_base.URLFeaturizer method), 1126
fit_transform() (evalml.pipelines.components.transformers.URLFeaturizer method), 1178
fit_transform() (evalml.pipelines.components.transformers.ReplaceNullableTypes method), 1272
fit_transform() (evalml.pipelines.components.transformers.RFClassifierRFE method), 1275
fit_transform() (evalml.pipelines.components.transformers.RFClassifierSelectFromModel method), 1278
fit_transform() (evalml.pipelines.components.transformers.RFRegressorRFE method), 1281
fit_transform() (evalml.pipelines.components.transformers.RFRegressorSelectFromModel method), 1284
fit_transform() (evalml.pipelines.components.samplers.BaseSampler method), 1181
fit_transform() (evalml.pipelines.components.samplers.OneHotSampler method), 1191
fit_transform() (evalml.pipelines.components.transformers.samplers.OverSampler method), 1185
fit_transform() (evalml.pipelines.components.transformers.samplers.UnderSampler method), 1194
fit_transform() (evalml.pipelines.components.transformers.samplers.underSampler method), 1188
fit_transform() (evalml.pipelines.components.transformers.NaturalLanguageFeaturizer method), 1197
fit_transform() (evalml.pipelines.components.transformers.scalers.StandardScaler method), 1199
fit_transform() (evalml.pipelines.components.PolynomialDecomposer method), 1286
fit_transform() (evalml.pipelines.components.transformers.SelectColumns method), 1289
fit_transform() (evalml.pipelines.components.transformers.SelectByType method), 1291
fit_transform() (evalml.pipelines.components.transformers.SimpleImputer method), 1293
fit_transform() (evalml.pipelines.components.transformers.StandardScaler method), 1297
fit_transform() (evalml.pipelines.components.transformers.STLDecomposer method), 1302
fit_transform() (evalml.pipelines.components.transformers.TargetEncoder method), 1305
fit_transform() (evalml.pipelines.components.transformers.TargetImputer method), 1308
fit_transform() (evalml.pipelines.components.transformers.TimeSeriesFeaturizer method), 1311
fit_transform() (evalml.pipelines.components.transformers.TimeSeriesFeaturizer method), 1314
fit_transform() (evalml.pipelines.components.transformers.TimeSeriesFeaturizer method), 1316
fit_transform() (evalml.pipelines.components.transformers.transformer method), 1212
fit_transform() (evalml.pipelines.components.transformers.transformer_base.EmptyFeatureUnderSampler method), 1320
fit_transform() (evalml.pipelines.components.transformers.transformer_base.URLFeaturizer method), 1322
fit_transform() (evalml.pipelines.components.Undersampler method), 1534
fit_transform() (evalml.pipelines.components.URLFeaturizer method), 1536
fit_transform() (evalml.pipelines.DFSTransformer method), 1680
fit_transform() (evalml.pipelines.DropNaNRowsTransformer method), 1682
fit_transform() (evalml.pipelines.FeatureSelector method), 1706
fit_transform() (evalml.pipelines.Imputer method), 1708
fit_transform() (evalml.pipelines.multiclass_classification_pipeline.MulticlassClassificationPipeline method), 1577
fit_transform() (evalml.pipelines.MulticlassClassificationPipeline

method), 1729
fit_transform() (evalml.pipelines.OneHotEncoder method), 1735
fit_transform() (evalml.pipelines.OrdinalEncoder method), 1738
fit_transform() (evalml.pipelines.PerColumnImputer method), 1741
fit_transform() (evalml.pipelines.pipeline_base.PipelineBase method), 1584
fit_transform() (evalml.pipelines.PipelineBase method), 1744
fit_transform() (evalml.pipelines.regression_pipeline.RegressionPipeline method), 1592
fit_transform() (evalml.pipelines.RegressionPipeline method), 1760
fit_transform() (evalml.pipelines.RFClassifierSelectFromModel method), 1765
fit_transform() (evalml.pipelines.RFRegressorSelectFromModel method), 1768
fit_transform() (evalml.pipelines.SimpleImputer method), 1771
fit_transform() (evalml.pipelines.StandardScaler method), 1784
fit_transform() (evalml.pipelines.TargetEncoder method), 1793
fit_transform() (evalml.pipelines.time_series_classification_pipelines.TimeSeriesBinaryClassificationPipeline method), 1599
fit_transform() (evalml.pipelines.time_series_classification_pipelines.TimeSeriesClassificationPipeline method), 1607
fit_transform() (evalml.pipelines.time_series_classification_pipelines.TimeSeriesMulticlassClassificationPipeline method), 1615
fit_transform() (evalml.pipelines.time_series_pipeline_base.TimeSeriesPipelineBase method), 1623
fit_transform() (evalml.pipelines.time_series_regression_pipeline.TimeSeriesRegressionPipeline method), 1630
fit_transform() (evalml.pipelines.TimeSeriesBinaryClassificationPipeline method), 1798
fit_transform() (evalml.pipelines.TimeSeriesClassificationPipeline method), 1805
fit_transform() (evalml.pipelines.TimeSeriesFeaturizer method), 1811
fit_transform() (evalml.pipelines.TimeSeriesImputer method), 1814
fit_transform() (evalml.pipelines.TimeSeriesMulticlassClassificationPipeline method), 1818
fit_transform() (evalml.pipelines.TimeSeriesRegressionPipeline method), 1826
fit_transform() (evalml.pipelines.TimeSeriesRegularizer method), 1833
fit_transform() (evalml.pipelines.Transformer method), 1836
force_plot() (in module evalml.model_understanding.force_plots), 412
FraudCost (class in evalml.objectives), 563
FraudCost (class in evalml.objectives.fraud_cost), 446
full_rankings (evalml.automl.automl_search.AutoMLSearch property), 262
full_rankings (evalml.automl.AutoMLSearch property), 278
full_rankings (evalml.AutoMLSearch property), 1918


## G


generate_component_code() (in module evalml.pipelines.components.utils), 1328
generate_order() (evalml.pipelines.component_graph.ComponentGraph class method), 1571
generate_order() (evalml.pipelines.ComponentGraph class method), 1668
generate_pipeline_code() (in module evalml.pipelines.utils), 1636
generate_pipeline_example() (in module evalml.pipelines.utils), 1637
get_action_from_defaults() (evalml.data_checks.data_check_action_option.DataCheckAction method), 293
get_action_from_defaults() (evalml.data_checks.DataCheckActionOption method), 347
get_actions_from_option_defaults() (in module evalml.pipelines.utils), 1637
get_all_objective_names() (in module evalml.objectives), 566
get_all_objective_names() (in module evalml.objectives.utils), 530
get_best_sampler_for_data() (in module evalml.automl.utils), 270
get_boxplot_data() (evalml.data_checks.outliers_data_check.OutliersD static method), 326
get_boxplot_data() (evalml.data_checks.OutliersDataCheck static method), 376
get_component() (evalml.pipelines.binary_classification_pipeline.Binary method), 1557
get_component() (evalml.pipelines.BinaryClassificationPipeline method), 1649
get_component() (evalml.pipelines.classification_pipeline.ClassificationP method), 1565
get_component() (evalml.pipelines.ClassificationPipeline method), 1662
get_component() (evalml.pipelines.component_graph.ComponentGraph method), 1571
get_component() (evalml.pipelines.ComponentGraph method), 1668
get_component() (evalml.pipelines.multiclass_classification_pipeline.Mu method), 1578
get_component() (evalml.pipelines.MulticlassClassificationPipeline method), 1730

get_component() (*evalml.pipelines.pipeline_base.PipelineBase method*), 1584
get_component() (*evalml.pipelines.PipelineBase method*), 1744
get_component() (*evalml.pipelines.regression_pipeline.RegressionPipeline method*), 1592
get_component() (*evalml.pipelines.RegressionPipeline method*), 1760
get_component() (*evalml.pipelines.time_series_classification_pipeline.TimeSeriesBinaryClassificationPipeline method*), 1600
get_component() (*evalml.pipelines.time_series_classification_pipeline.TimeSeriesMulticlassClassificationPipeline method*), 1607
get_component() (*evalml.pipelines.time_series_classification_pipeline.TimeSeriesClassificationPipeline method*), 1615
get_component() (*evalml.pipelines.time_series_pipeline_base.TimeSeriesPipelineBase method*), 1623
get_component() (*evalml.pipelines.time_series_regression_pipeline.TimeSeriesRegressionPipeline method*), 1630
get_component() (*evalml.pipelines.TimeSeriesBinaryClassificationPipeline method*), 1798
get_component() (*evalml.pipelines.TimeSeriesClassificationPipeline method*), 1805
get_component() (*evalml.pipelines.TimeSeriesMulticlassClassificationPipeline method*), 1818
get_component() (*evalml.pipelines.TimeSeriesRegressionPipeline method*), 1826
get_component_input_logical_types() (*evalml.pipelines.component_graph.ComponentGraph method*), 1571
get_component_input_logical_types() (*evalml.pipelines.ComponentGraph method*), 1668
get_core_objective_names() (*in module evalml.objectives*), 566
get_core_objective_names() (*in module evalml.objectives.utils*), 530
get_core_objectives() (*in module evalml.objectives*), 566
get_core_objectives() (*in module evalml.objectives.utils*), 531
get_default_primary_search_objective() (*in module evalml.automl*), 280
get_default_primary_search_objective() (*in module evalml.automl.utils*), 271
get_ensembler_input_pipelines() (*evalml.automl.automl_search.AutoMLSearch method*), 262
get_ensembler_input_pipelines() (*evalml.automl.AutoMLSearch method*), 278
get_ensembler_input_pipelines() (*evalml.AutoMLSearch method*), 1918
get_estimators() (*evalml.pipelines.component_graph.ComponentGraph method*), 1572
get_estimators() (*evalml.pipelines.ComponentGraph method*), 1669
get_estimators() (*in module evalml.pipelines.components.utils*), 1329
get_evalml_black_config() (*in module evalml.utils.cli_utils*), 1899
get_evalml_pip_requirements() (*in module evalml.utils.cli_utils*), 1899
get_evalml_requirements() (*in module evalml.utils.cli_utils*), 1899
get_evalml_root() (*in module evalml.utils.cli_utils*), 1899
get_feature_names() (*evalml.pipelines.components.DateTimeFeaturizer method*), 1158
get_feature_names() (*evalml.pipelines.components.DelayedFeatureTransformer method*), 1439
get_feature_names() (*evalml.pipelines.components.OneHotEncoder method*), 1442
get_feature_names() (*evalml.pipelines.components.OrdinalEncoder method*), 1513
get_feature_names() (*evalml.pipelines.components.TargetEncoder method*), 1217
get_feature_names() (*evalml.pipelines.components.transformers.DateTimeFeaturizer method*), 974
get_feature_names() (*evalml.pipelines.components.transformers.encoders.onehot_encoder method*), 989
get_feature_names() (*evalml.pipelines.components.transformers.encoders.OneHotEncoder method*), 978
get_feature_names() (*evalml.pipelines.components.transformers.encoders.ordinal_encoder method*), 992
get_feature_names() (*evalml.pipelines.components.transformers.encoders.OrdinalEncoder method*), 983
get_feature_names() (*evalml.pipelines.components.transformers.encoders.target_encoder method*), 995
get_feature_names() (*evalml.pipelines.components.transformers.encoders.TargetEncoder method*), 1253
get_feature_names() (*evalml.pipelines.components.transformers.OneHotEncoder method*), 1257
get_feature_names() (*evalml.pipelines.components.transformers.OrdinalEncoder method*), 1070
get_feature_names() (*evalml.pipelines.components.transformers.preprocessing.datetimefeaturizer.DateTimeFeaturizer method*), 1070

get_feature_names() (evalml.pipelines.components.transformers.preprocessing.DateTimeFeaturizer method), 1130
get_feature_names() (evalml.pipelines.components.transformers.TargetEncoder method), 1302
get_feature_names() (evalml.pipelines.OneHotEncoder method), 1735
get_feature_names() (evalml.pipelines.OrdinalEncoder method), 1738
get_feature_names() (evalml.pipelines.TargetEncoder method), 1793
get_forecast_period() (evalml.pipelines.time_series_regression_pipeline.TimeSeriesRegressionPipeline method), 1630
get_forecast_period() (evalml.pipelines.TimeSeriesRegressionPipeline method), 1826
get_forecast_predictions() (evalml.pipelines.time_series_regression_pipeline.TimeSeriesRegressionPipeline method), 1631
get_forecast_predictions() (evalml.pipelines.TimeSeriesRegressionPipeline method), 1827
get_hyperparameter_ranges() (evalml.pipelines.binary_classification_pipeline.BinaryClassificationPipeline method), 1557
get_hyperparameter_ranges() (evalml.pipelines.BinaryClassificationPipeline method), 1649
get_hyperparameter_ranges() (evalml.pipelines.classification_pipeline.ClassificationPipeline method), 1565
get_hyperparameter_ranges() (evalml.pipelines.ClassificationPipeline method), 1662
get_hyperparameter_ranges() (evalml.pipelines.multiclass_classification_pipeline.MulticlassClassificationPipeline method), 1578
get_hyperparameter_ranges() (evalml.pipelines.MulticlassClassificationPipeline method), 1730
get_hyperparameter_ranges() (evalml.pipelines.pipeline_base.PipelineBase method), 1584
get_hyperparameter_ranges() (evalml.pipelines.PipelineBase method), 1744
get_hyperparameter_ranges() (evalml.pipelines.regression_pipeline.RegressionPipeline method), 1593
get_hyperparameter_ranges() (evalml.pipelines.RegressionPipeline method), 1761
get_hyperparameter_ranges() (evalml.pipelines.time_series_classification_pipelines.TimeSeries method), 1600
get_hyperparameter_ranges() (evalml.pipelines.time_series_classification_pipelines.TimeSeries method), 1607
get_hyperparameter_ranges() (evalml.pipelines.time_series_classification_pipelines.TimeSeries method), 1615
get_hyperparameter_ranges() (evalml.pipelines.time_series_pipeline_base.TimeSeriesPipelineB method), 1623
get_hyperparameter_ranges() (evalml.pipelines.time_series_regression_pipeline.TimeSeriesReg method), 1631
get_hyperparameter_ranges() (evalml.pipelines.TimeSeriesBinaryClassificationPipeline method), 1798
get_hyperparameter_ranges() (evalml.pipelines.TimeSeriesClassificationPipeline method), 1805
get_hyperparameter_ranges() (evalml.pipelines.TimeSeriesMulticlassClassificationPipeline method), 1819
get_hyperparameter_ranges() (evalml.pipelines.TimeSeriesRegressionPipeline method), 1827
get_importable_subclasses() (in module evalml.utils), 1911
get_importable_subclasses() (in module evalml.utils.gen_utils), 1903
get_influential_features() (in module evalml.model_understanding.feature_explanations), 410
get_inputs() (evalml.pipelines.component_graph.ComponentGraph method), 1572
get_inputs() (evalml.pipelines.ComponentGraph method), 1668
get_installed_packages() (in module evalml.utils.cli_utils), 1900
get_last_component() (evalml.pipelines.component_graph.ComponentGraph method), 1572
get_last_component() (evalml.pipelines.ComponentGraph method), 1669
get_linear_coefficients() (in module evalml.model_understanding), 432
get_linear_coefficients() (in module evalml.model_understanding.visualizations), 422

get_logger() (*in module evalml.utils*), 1911
get_logger() (*in module evalml.utils.logger*), 1906
get_n_splits() (*evalml.preprocessing.data_splitters.KFold method*), 1860
get_n_splits() (*evalml.preprocessing.data_splitters.no_split.NoSplit static method*), 1853
get_n_splits() (*evalml.preprocessing.data_splitters.NoSplit static method*), 1861
get_n_splits() (*evalml.preprocessing.data_splitters.sk_splitter.KFold method*), 1854
get_n_splits() (*evalml.preprocessing.data_splitters.sk_splitter.StratifiedKFold method*), 1855
get_n_splits() (*evalml.preprocessing.data_splitters.StratifiedKFold method*), 1862
get_n_splits() (*evalml.preprocessing.data_splitters.time_series_split.TimeSeriesSplit method*), 1857
get_n_splits() (*evalml.preprocessing.data_splitters.TimeSeriesSplit method*), 1864
get_n_splits() (*evalml.preprocessing.data_splitters.training_validation_split.TrainingValidationSplit static method*), 1859
get_n_splits() (*evalml.preprocessing.data_splitters.TrainingValidationSplit static method*), 1865
get_n_splits() (*evalml.preprocessing.NoSplit static method*), 1870
get_n_splits() (*evalml.preprocessing.TimeSeriesSplit method*), 1873
get_n_splits() (*evalml.preprocessing.TrainingValidationSplit static method*), 1875
get_names() (*evalml.pipelines.components.FeatureSelector method*), 1403
get_names() (*evalml.pipelines.components.RFClassifierRFESelector method*), 1470
get_names() (*evalml.pipelines.components.RFClassifierSelectFromModel method*), 1473
get_names() (*evalml.pipelines.components.RFRegressorRFESelector method*), 1476
get_names() (*evalml.pipelines.components.RFRegressorSelectFromModel method*), 1479
get_names() (*evalml.pipelines.components.transformers.feature_selection.base.FeatureSelection method*), 998
get_names() (*evalml.pipelines.components.transformers.feature_selection.FeatureSelector method*), 1018
get_names() (*evalml.pipelines.components.transformers.feature_selection.recursive_feature_elimination_selector.RecursiveFeatureEl method*), 1002
get_names() (*evalml.pipelines.components.transformers.feature_selection.recursive_feature_elimination_selector.RFClassifierRFESe method*), 1005
get_names() (*evalml.pipelines.components.transformers.feature_selection.recursive_feature_elimination_selector.RFRegressorRFESe method*), 1008
get_names() (*evalml.pipelines.components.transformers.feature_selection.rf_classifier_feature_selector.RFClassifierSelectFromMode method*), 1012
get_names() (*evalml.pipelines.components.transformers.feature_selection.rf_regressor_feature_selector.RFRegressorSelectFromMod method*), 1015
get_names() (*evalml.pipelines.components.transformers.feature_selection.RFClassifierRFESelector method*), 1022
get_names() (*evalml.pipelines.components.transformers.feature_selection method*), 1025
get_names() (*evalml.pipelines.components.transformers.feature_selection method*), 1028
get_names() (*evalml.pipelines.components.transformers.feature_selection method*), 1031
get_names() (*evalml.pipelines.components.transformers.FeatureSelector method*), 1235
get_names() (*evalml.pipelines.components.transformers.RFClassifierRFE method*), 1275
get_names() (*evalml.pipelines.components.transformers.RFClassifierSele method*), 1278
get_names() (*evalml.pipelines.components.transformers.RFRegressorRFE method*), 1281
get_names() (*evalml.pipelines.components.transformers.RFRegressorSele method*), 1284
get_names() (*evalml.pipelines.FeatureSelector method*), 1706
get_names() (*evalml.pipelines.RFClassifierSelectFromModel method*), 1765
get_names() (*evalml.pipelines.RFRegressorSelectFromModel method*), 1768
get_non_core_objectives() (*in module evalml.objectives*), 566
get_non_core_objectives() (*in module evalml.objectives.utils*), 531
get_null_column_information() (*evalml.data_checks.null_data_check.NullDataCheck static method*), 322
get_null_column_information() (*evalml.data_checks.NullDataCheck static method*), 372
get_null_row_information() (*evalml.data_checks.null_data_check.NullDataCheck static method*), 322
get_null_row_information() (*evalml.data_checks.NullDataCheck static method*), 372
get_objective() (*in module evalml.objectives*), 567
get_objective() (*in module evalml.objectives.utils*), 531
get_optimization_objectives() (*in module evalml.objectives*), 567
get_optimization_objectives() (*in module evalml.objectives.utils*), 532
get_params() (*evalml.pipelines.components.estimators.ProphetRegressor method*), 924
get_params() (*evalml.pipelines.components.estimators.regressors.prophet method*), 781
get_params() (*evalml.pipelines.components.estimators.regressors.Prophe method*), 836
get_params() (*evalml.pipelines.components.ProphetRegressor method*), 1458
get_params() (*evalml.pipelines.components.utils.WrappedSKClassifier*

*method*), 1331
get_params() (*evalml.pipelines.components.utils.WrappedSKRegressor method*), 1333
get_params() (*evalml.pipelines.ProphetRegressor method*), 1750
get_pipeline() (*evalml.automl.automl_search.AutoMLSearch method*), 262
get_pipeline() (*evalml.automl.AutoMLSearch method*), 278
get_pipeline() (*evalml.AutoMLSearch method*), 1918
get_pipelines_from_component_graphs() (*in module evalml.automl.utils*), 271
get_prediction_intervals() (*evalml.pipelines.ARIMARegressor method*), 1644
get_prediction_intervals() (*evalml.pipelines.CatBoostClassifier method*), 1654
get_prediction_intervals() (*evalml.pipelines.CatBoostRegressor method*), 1657
get_prediction_intervals() (*evalml.pipelines.components.ARIMARegressor method*), 1339
get_prediction_intervals() (*evalml.pipelines.components.BaselineClassifier method*), 1342
get_prediction_intervals() (*evalml.pipelines.components.BaselineRegressor method*), 1345
get_prediction_intervals() (*evalml.pipelines.components.CatBoostClassifier method*), 1349
get_prediction_intervals() (*evalml.pipelines.components.CatBoostRegressor method*), 1352
get_prediction_intervals() (*evalml.pipelines.components.DecisionTreeClassifier method*), 1361
get_prediction_intervals() (*evalml.pipelines.components.DecisionTreeRegressor method*), 1365
get_prediction_intervals() (*evalml.pipelines.components.ElasticNetClassifier method*), 1381
get_prediction_intervals() (*evalml.pipelines.components.ElasticNetRegressor method*), 1384
get_prediction_intervals() (*evalml.pipelines.components.ensemble.stacked_ensemble_base.StackedEnsembleBase method*), 621
get_prediction_intervals() (*evalml.pipelines.components.ensemble.stacked_ensemble_classifier.StackedEnsembleClassifier method*), 625
get_prediction_intervals() (*evalml.pipelines.components.ensemble.stacked_ensemble_regres method*), 629
get_prediction_intervals() (*evalml.pipelines.components.ensemble.StackedEnsembleBase method*), 633
get_prediction_intervals() (*evalml.pipelines.components.ensemble.StackedEnsembleClassifie method*), 637
get_prediction_intervals() (*evalml.pipelines.components.ensemble.StackedEnsembleRegress method*), 641
get_prediction_intervals() (*evalml.pipelines.components.Estimator method*), 1389
get_prediction_intervals() (*evalml.pipelines.components.estimators.ARIMARegressor method*), 861
get_prediction_intervals() (*evalml.pipelines.components.estimators.BaselineClassifier method*), 864
get_prediction_intervals() (*evalml.pipelines.components.estimators.BaselineRegressor method*), 867
get_prediction_intervals() (*evalml.pipelines.components.estimators.CatBoostClassifier method*), 871
get_prediction_intervals() (*evalml.pipelines.components.estimators.CatBoostRegressor method*), 874
get_prediction_intervals() (*evalml.pipelines.components.estimators.classifiers.baseline_clas method*), 645
get_prediction_intervals() (*evalml.pipelines.components.estimators.classifiers.BaselineClass method*), 698
get_prediction_intervals() (*evalml.pipelines.components.estimators.classifiers.catboost_clas method*), 649
get_prediction_intervals() (*evalml.pipelines.components.estimators.classifiers.CatBoostClas method*), 701
get_prediction_intervals() (*evalml.pipelines.components.estimators.classifiers.decision_tree_ method*), 653
get_prediction_intervals() (*evalml.pipelines.components.estimators.classifiers.DecisionTree method*), 705
get_prediction_intervals() (*evalml.pipelines.components.estimators.classifiers.elasticnet_cla method*), 657
get_prediction_intervals() (*evalml.pipelines.components.estimators.classifiers.ElasticNetCla method*), 708

get_prediction_intervals() (evalml.pipelines.components.estimators.classifiers.et_classifier.ExtraTreesClassifier method), 661
get_prediction_intervals() (evalml.pipelines.components.estimators.classifiers.ExtraTreesClassifier method), 712
get_prediction_intervals() (evalml.pipelines.components.estimators.classifiers.kneighbors_classifier.KNeighborsClassifier method), 665
get_prediction_intervals() (evalml.pipelines.components.estimators.classifiers.KNeighborsClassifier method), 716
get_prediction_intervals() (evalml.pipelines.components.estimators.classifiers.lightgbm_classifier.LightGBMClassifier method), 669
get_prediction_intervals() (evalml.pipelines.components.estimators.classifiers.LightGBMClassifier method), 719
get_prediction_intervals() (evalml.pipelines.components.estimators.classifiers.logistic_regression_classifier.LogisticRegressionClassifier method), 673
get_prediction_intervals() (evalml.pipelines.components.estimators.classifiers.LogisticRegressionClassifier method), 723
get_prediction_intervals() (evalml.pipelines.components.estimators.classifiers.RandomForestClassifier method), 726
get_prediction_intervals() (evalml.pipelines.components.estimators.classifiers.rf_classifier.RandomForestClassifier method), 677
get_prediction_intervals() (evalml.pipelines.components.estimators.classifiers.svm_classifier.SVMClassifier method), 680
get_prediction_intervals() (evalml.pipelines.components.estimators.classifiers.SVMClassifier method), 729
get_prediction_intervals() (evalml.pipelines.components.estimators.classifiers.vowpal_wabbit_binary_classifier.VowpalWabbitBinaryClassifier method), 684
get_prediction_intervals() (evalml.pipelines.components.estimators.classifiers.vowpal_wabbit_multi_class_classifier.VowpalWabbitMultiClassClassifier method), 687
get_prediction_intervals() (evalml.pipelines.components.estimators.classifiers.VowpalWabbitBinaryClassifier method), 691
get_prediction_intervals() (evalml.pipelines.components.estimators.classifiers.VowpalWabbitMultiClassClassifier method), 733
get_prediction_intervals() (evalml.pipelines.components.estimators.classifiers.xgboost_classifier.XGBoostClassifier method), 736
get_prediction_intervals() (evalml.pipelines.components.estimators.classifiers.XGBoostClassifier method), 694
get_prediction_intervals() (evalml.pipelines.components.estimators.classifiers.XGBoostClassifier method), 739
get_prediction_intervals() (evalml.pipelines.components.estimators.DecisionTreeClassifier method), 878
get_prediction_intervals() (evalml.pipelines.components.estimators.DecisionTreeRegressor method), 882
get_prediction_intervals() (evalml.pipelines.components.estimators.ElasticNetClassifier method), 885
get_prediction_intervals() (evalml.pipelines.components.estimators.ElasticNetRegressor method), 888
get_prediction_intervals() (evalml.pipelines.components.estimators.Estimator method), 891
get_prediction_intervals() (evalml.pipelines.components.estimators.Estimator method), 856
get_prediction_intervals() (evalml.pipelines.components.estimators.ExponentialSmoothingR method), 895
get_prediction_intervals() (evalml.pipelines.components.estimators.ExtraTreesClassifier method), 899
get_prediction_intervals() (evalml.pipelines.components.estimators.ExtraTreesRegressor method), 903
get_prediction_intervals() (evalml.pipelines.components.estimators.KNeighborsClassifier method), 906
get_prediction_intervals() (evalml.pipelines.components.estimators.LightGBMClassifier method), 910
get_prediction_intervals() (evalml.pipelines.components.estimators.LightGBMRegressor method), 914
get_prediction_intervals() (evalml.pipelines.components.estimators.VowpalWabbitBinaryClassifier method), 917
get_prediction_intervals() (evalml.pipelines.components.estimators.VowpalWabbitMultiClassClassifier method), 920
get_prediction_intervals() (evalml.pipelines.components.estimators.ProphetRegressor method), 924
get_prediction_intervals() (evalml.pipelines.components.estimators.RandomForestClassifier method), 927
get_prediction_intervals() (evalml.pipelines.components.estimators.RandomForestRegressor method), 930

get_prediction_intervals() (evalml.pipelines.components.estimators.regressors.arima_regressor.ARIMARegressor method), 745
get_prediction_intervals() (evalml.pipelines.components.estimators.regressors.ARIMARegressor method), 804
get_prediction_intervals() (evalml.pipelines.components.estimators.regressors.baseline_regressor.BaselineRegressor method), 748
get_prediction_intervals() (evalml.pipelines.components.estimators.regressors.BaselineRegressor method), 807
get_prediction_intervals() (evalml.pipelines.components.estimators.regressors.catboost_regressor.CatBoostRegressor method), 752
get_prediction_intervals() (evalml.pipelines.components.estimators.regressors.CatBoostRegressor method), 810
get_prediction_intervals() (evalml.pipelines.components.estimators.regressors.decision_tree_regressor.DecisionTreeRegressor method), 756
get_prediction_intervals() (evalml.pipelines.components.estimators.regressors.DecisionTreeRegressor method), 814
get_prediction_intervals() (evalml.pipelines.components.estimators.regressors.elasticnet_regressor.ElasticNetRegressor method), 759
get_prediction_intervals() (evalml.pipelines.components.estimators.regressors.ElasticNetRegressor method), 817
get_prediction_intervals() (evalml.pipelines.components.estimators.regressors.et_regressor.ExtraTreesRegressor method), 764
get_prediction_intervals() (evalml.pipelines.components.estimators.regressors.exponential_smoothing_regressor.ExponentialSmoothingRegressor method), 768
get_prediction_intervals() (evalml.pipelines.components.estimators.regressors.ExponentialSmoothingRegressor method), 821
get_prediction_intervals() (evalml.pipelines.components.estimators.regressors.ExtraTreesRegressor method), 825
get_prediction_intervals() (evalml.pipelines.components.estimators.regressors.lightgbm_regressor.LightGBMRegressor method), 772
get_prediction_intervals() (evalml.pipelines.components.estimators.regressors.LightGBMRegressor method), 829
get_prediction_intervals() (evalml.pipelines.components.estimators.regressors.linear_regressor.LinearRegressor method), 775
get_prediction_intervals() (evalml.pipelines.components.estimators.regressors.LinearRegressor method), 832
get_prediction_intervals() (evalml.pipelines.components.estimators.regressors.prophet_regressor.ProphetRegressor method), 781
get_prediction_intervals() (evalml.pipelines.components.estimators.regressors.ProphetRegressor method), 836
get_prediction_intervals() (evalml.pipelines.components.estimators.regressors.RandomForestRegressor method), 839
get_prediction_intervals() (evalml.pipelines.components.estimators.regressors.rf_regressor.RandomForestRegressor method), 785
get_prediction_intervals() (evalml.pipelines.components.estimators.regressors.svm_regressor.SVMRegressor method), 788
get_prediction_intervals() (evalml.pipelines.components.estimators.regressors.SVMRegressor method), 842
get_prediction_intervals() (evalml.pipelines.components.estimators.regressors.time_series_baseline_regressor.TimeSeriesBaselineRegressor method), 792
get_prediction_intervals() (evalml.pipelines.components.estimators.regressors.TimeSeriesBaselineRegressor method), 845
get_prediction_intervals() (evalml.pipelines.components.estimators.regressors.vowpal_wabbit_regressor.VowpalWabbitRegressor method), 795
get_prediction_intervals() (evalml.pipelines.components.estimators.regressors.VowpalWabbitRegressor method), 848
get_prediction_intervals() (evalml.pipelines.components.estimators.regressors.xgboost_regressor.XGBoostRegressor method), 799
get_prediction_intervals() (evalml.pipelines.components.estimators.regressors.XGBoostRegressor method), 852
get_prediction_intervals() (evalml.pipelines.components.estimators.SVMClassifier method), 933
get_prediction_intervals() (evalml.pipelines.components.estimators.SVMRegressor method), 936
get_prediction_intervals() (evalml.pipelines.components.estimators.TimeSeriesBaselineEstimator method), 939
get_prediction_intervals() (evalml.pipelines.components.estimators.VowpalWabbitBinaryClassifier method), 943
get_prediction_intervals() (evalml.pipelines.components.estimators.VowpalWabbitMulticlassClassifier method), 946
get_prediction_intervals() (evalml.pipelines.components.estimators.VowpalWabbitRegressor method), 949

get_prediction_intervals() (*evalml.pipelines.components.estimators.XGBoostClassifier method*), 952
get_prediction_intervals() (*evalml.pipelines.components.estimators.XGBoostRegressor method*), 955
get_prediction_intervals() (*evalml.pipelines.components.ExponentialSmoothingRegressor method*), 1393
get_prediction_intervals() (*evalml.pipelines.components.ExtraTreesClassifier method*), 1397
get_prediction_intervals() (*evalml.pipelines.components.ExtraTreesRegressor method*), 1400
get_prediction_intervals() (*evalml.pipelines.components.KNeighborsClassifier method*), 1409
get_prediction_intervals() (*evalml.pipelines.components.LightGBMClassifier method*), 1415
get_prediction_intervals() (*evalml.pipelines.components.LightGBMRegressor method*), 1419
get_prediction_intervals() (*evalml.pipelines.components.LinearRegressor method*), 1424
get_prediction_intervals() (*evalml.pipelines.components.LogisticRegressionClassifier method*), 1428
get_prediction_intervals() (*evalml.pipelines.components.ProphetRegressor method*), 1458
get_prediction_intervals() (*evalml.pipelines.components.RandomForestClassifier method*), 1461
get_prediction_intervals() (*evalml.pipelines.components.RandomForestRegressor method*), 1464
get_prediction_intervals() (*evalml.pipelines.components.StackedEnsembleBase method*), 1489
get_prediction_intervals() (*evalml.pipelines.components.StackedEnsembleClassifier method*), 1493
get_prediction_intervals() (*evalml.pipelines.components.StackedEnsembleRegressor method*), 1496
get_prediction_intervals() (*evalml.pipelines.components.SVMClassifier method*), 1507
get_prediction_intervals() (*evalml.pipelines.components.SVMRegressor method*), 1510
get_prediction_intervals() (*evalml.pipelines.components.TimeSeriesBaselineEstimator method*), 1519
get_prediction_intervals() (*evalml.pipelines.components.VowpalWabbitBinaryClassifier method*), 1539
get_prediction_intervals() (*evalml.pipelines.components.VowpalWabbitMulticlassClassifier method*), 1542
get_prediction_intervals() (*evalml.pipelines.components.VowpalWabbitRegressor method*), 1545
get_prediction_intervals() (*evalml.pipelines.components.XGBoostClassifier method*), 1548
get_prediction_intervals() (*evalml.pipelines.components.XGBoostRegressor method*), 1551
get_prediction_intervals() (*evalml.pipelines.DecisionTreeClassifier method*), 1673
get_prediction_intervals() (*evalml.pipelines.DecisionTreeRegressor method*), 1677
get_prediction_intervals() (*evalml.pipelines.ElasticNetClassifier method*), 1685
get_prediction_intervals() (*evalml.pipelines.ElasticNetRegressor method*), 1688
get_prediction_intervals() (*evalml.pipelines.Estimator method*), 1692
get_prediction_intervals() (*evalml.pipelines.ExponentialSmoothingRegressor method*), 1695
get_prediction_intervals() (*evalml.pipelines.ExtraTreesClassifier method*), 1699
get_prediction_intervals() (*evalml.pipelines.ExtraTreesRegressor method*), 1703
get_prediction_intervals() (*evalml.pipelines.KNeighborsClassifier method*), 1712
get_prediction_intervals() (*evalml.pipelines.LightGBMClassifier method*), 1715
get_prediction_intervals() (*evalml.pipelines.LightGBMRegressor method*), 1719
get_prediction_intervals() (*evalml.pipelines.LinearRegressor method*), 1721
get_prediction_intervals()

(*evalml.pipelines.LogisticRegressionClassifier method*), 1725
get_prediction_intervals() (*evalml.pipelines.ProphetRegressor method*), 1750
get_prediction_intervals() (*evalml.pipelines.RandomForestClassifier method*), 1753
get_prediction_intervals() (*evalml.pipelines.RandomForestRegressor method*), 1756
get_prediction_intervals() (*evalml.pipelines.StackedEnsembleBase method*), 1774
get_prediction_intervals() (*evalml.pipelines.StackedEnsembleClassifier method*), 1778
get_prediction_intervals() (*evalml.pipelines.StackedEnsembleRegressor method*), 1781
get_prediction_intervals() (*evalml.pipelines.SVMClassifier method*), 1787
get_prediction_intervals() (*evalml.pipelines.SVMRegressor method*), 1790
get_prediction_intervals() (*evalml.pipelines.time_series_regression_pipeline.TimeSeriesRegressionPipeline method*), 1631
get_prediction_intervals() (*evalml.pipelines.TimeSeriesRegressionPipeline method*), 1827
get_prediction_intervals() (*evalml.pipelines.VowpalWabbitBinaryClassifier method*), 1839
get_prediction_intervals() (*evalml.pipelines.VowpalWabbitMulticlassClassifier method*), 1842
get_prediction_intervals() (*evalml.pipelines.VowpalWabbitRegressor method*), 1845
get_prediction_intervals() (*evalml.pipelines.XGBoostClassifier method*), 1848
get_prediction_intervals() (*evalml.pipelines.XGBoostRegressor method*), 1851
get_prediction_intevals_for_tree_regressors() (*in module evalml.pipelines.components.utils*), 1329
get_prediction_vs_actual_data() (*in module evalml.model_understanding*), 432
get_prediction_vs_actual_data() (*in module evalml.model_understanding.visualizations*), 422
get_prediction_vs_actual_over_time_data() (*in module evalml.model_understanding*), 432
get_prediction_vs_actual_over_time_data() (*in module evalml.model_understanding.visualizations*), 423
get_random_seed() (*in module evalml.utils*), 1911
get_random_seed() (*in module evalml.utils.gen_utils*), 1904
get_random_state() (*in module evalml.utils*), 1911
get_random_state() (*in module evalml.utils.gen_utils*), 1904
get_ranking_objectives() (*in module evalml.objectives*), 567
get_ranking_objectives() (*in module evalml.objectives.utils*), 532
get_result() (*evalml.automl.engine.cf_engine.CFComputation method*), 239
get_result() (*evalml.automl.engine.dask_engine.DaskComputation method*), 242
get_result() (*evalml.automl.engine.engine_base.EngineComputation method*), 245
get_result() (*evalml.automl.engine.EngineComputation method*), 254
get_result() (*evalml.automl.engine.sequential_engine.SequentialComputation method*), 248
get_starting_parameters() (*evalml.tuners.grid_search_tuner.GridSearchTuner method*), 1885
get_starting_parameters() (*evalml.tuners.GridSearchTuner method*), 1892
get_starting_parameters() (*evalml.tuners.random_search_tuner.RandomSearchTuner method*), 1886
get_starting_parameters() (*evalml.tuners.RandomSearchTuner method*), 1894
get_starting_parameters() (*evalml.tuners.skopt_tuner.SKOptTuner method*), 1889
get_starting_parameters() (*evalml.tuners.SKOptTuner method*), 1896
get_starting_parameters() (*evalml.tuners.Tuner method*), 1897
get_starting_parameters() (*evalml.tuners.tuner.Tuner method*), 1890
get_sys_info() (*in module evalml.utils.cli_utils*), 1900
get_threshold_tuning_info() (*in module evalml.automl*), 280
get_threshold_tuning_info() (*in module evalml.automl.utils*), 271
get_time_index() (*in module evalml.utils*), 1912
get_time_index() (*in module evalml.utils.gen_utils*),

1904
get_trend_dataframe() (evalml.pipelines.components.PolynomialDecomposer method), 1454
get_trend_dataframe() (evalml.pipelines.components.STLDecomposer method), 1503
get_trend_dataframe() (evalml.pipelines.components.transformers.PolynomialDecomposer method), 1269
get_trend_dataframe() (evalml.pipelines.components.transformers.preprocessing.DateformatSplitter method), 1134
get_trend_dataframe() (evalml.pipelines.components.transformers.preprocessing.DelayedFeatureTransformer method), 1074
get_trend_dataframe() (evalml.pipelines.components.transformers.preprocessing.polynomial_decomposer.PolynomialDecomposer method), 1101
get_trend_dataframe() (evalml.pipelines.components.transformers.preprocessing.PolynomialDecomposer method), 1158
get_trend_dataframe() (evalml.pipelines.components.transformers.preprocessing.stl_decomposer.STLDecomposer method), 1110
get_trend_dataframe() (evalml.pipelines.components.transformers.preprocessing.STLDecomposer method), 1166
get_trend_dataframe() (evalml.pipelines.components.transformers.STLDecomposer method), 1298
get_trend_prediction_intervals() (evalml.pipelines.components.STLDecomposer method), 1503
get_trend_prediction_intervals() (evalml.pipelines.components.transformers.preprocessing.stl_decomposer.STLDecomposer method), 1110
get_trend_prediction_intervals() (evalml.pipelines.components.transformers.preprocessing.STLDecomposer method), 1166
get_trend_prediction_intervals() (evalml.pipelines.components.transformers.STLDecomposer method), 1298
Gini (class in evalml.objectives), 567
Gini (class in evalml.objectives.standard_metrics), 487
graph() (evalml.pipelines.binary_classification_pipeline.BinaryClassificationPipeline method), 1557
graph() (evalml.pipelines.BinaryClassificationPipeline method), 1649
graph() (evalml.pipelines.classification_pipeline.ClassificationPipeline method), 1565
graph() (evalml.pipelines.ClassificationPipeline method), 1662
graph() (evalml.pipelines.component_graph.ComponentGraph method), 1572
graph() (evalml.pipelines.ComponentGraph method), 1669
graph() (evalml.pipelines.multiclass_classification_pipeline.MulticlassClassificationPipeline method), 1578
graph() (evalml.pipelines.MulticlassClassificationPipeline method), 1730
graph() (evalml.pipelines.pipeline_base.PipelineBase method), 1585
graph() (evalml.pipelines.PipelineBase method), 1745
graph() (evalml.pipelines.regression_pipeline.RegressionPipeline method), 1593
graph() (evalml.pipelines.RegressionPipeline method), 1761
graph() (evalml.pipelines.time_series_classification_pipelines.TimeSeriesBinaryClassificationPipeline method), 1600
graph() (evalml.pipelines.time_series_classification_pipelines.TimeSeriesClassificationPipeline method), 1607
graph() (evalml.pipelines.time_series_classification_pipelines.TimeSeriesMulticlassClassificationPipeline method), 1615
graph() (evalml.pipelines.time_series_pipeline_base.TimeSeriesPipelineBase method), 1623
graph() (evalml.pipelines.time_series_regression_pipeline.TimeSeriesRegressionPipeline method), 1632
graph() (evalml.pipelines.TimeSeriesBinaryClassificationPipeline method), 1798
graph() (evalml.pipelines.TimeSeriesClassificationPipeline method), 1805
graph() (evalml.pipelines.TimeSeriesMulticlassClassificationPipeline method), 1819
graph() (evalml.pipelines.TimeSeriesRegressionPipeline method), 1828
graph_binary_objective_vs_threshold() (in module evalml.model_understanding), 433
graph_binary_objective_vs_threshold() (in module evalml.model_understanding.visualizations), 423
graph_confusion_matrix() (in module evalml.model_understanding), 433
graph_confusion_matrix() (in module evalml.model_understanding.metrics), 413
graph_dict() (evalml.pipelines.binary_classification_pipeline.BinaryClassificationPipeline method), 1557
graph_dict() (evalml.pipelines.BinaryClassificationPipeline method), 1649
graph_dict() (evalml.pipelines.classification_pipeline.ClassificationPipeline method), 1565
graph_dict() (evalml.pipelines.ClassificationPipeline method), 1662
graph_dict() (evalml.pipelines.multiclass_classification_pipeline.MulticlassClassificationPipeline method), 1578
graph_dict() (evalml.pipelines.MulticlassClassificationPipeline method), 1730
graph_dict() (evalml.pipelines.pipeline_base.PipelineBase

*method*), 1585
graph_dict() (*evalml.pipelines.PipelineBase method*), 1745
graph_dict() (*evalml.pipelines.regression_pipeline.RegressionPipeline method*), 1593
graph_dict() (*evalml.pipelines.RegressionPipeline method*), 1761
graph_dict() (*evalml.pipelines.time_series_classification.TimeSeriesBinaryClassificationPipeline method*), 1600
graph_dict() (*evalml.pipelines.time_series_classification_pipelines.TimeSeriesBinaryClassificationPipeline method*), 1608
graph_dict() (*evalml.pipelines.time_series_classification_pipelines.TimeSeriesMulticlassClassificationPipeline method*), 1615
graph_dict() (*evalml.pipelines.time_series_pipeline_base.TimeSeriesPipelineBase method*), 1623
graph_dict() (*evalml.pipelines.time_series_regression_pipeline.TimeSeriesRegressionPipeline method*), 1632
graph_dict() (*evalml.pipelines.TimeSeriesBinaryClassificationPipeline method*), 1798
graph_dict() (*evalml.pipelines.TimeSeriesClassificationPipeline method*), 1806
graph_dict() (*evalml.pipelines.TimeSeriesMulticlassClassificationPipeline method*), 1819
graph_dict() (*evalml.pipelines.TimeSeriesRegressionPipeline method*), 1828
graph_feature_importance() (*evalml.pipelines.binary_classification_pipeline.BinaryClassificationPipeline method*), 1557
graph_feature_importance() (*evalml.pipelines.BinaryClassificationPipeline method*), 1649
graph_feature_importance() (*evalml.pipelines.classification_pipeline.ClassificationPipeline method*), 1565
graph_feature_importance() (*evalml.pipelines.ClassificationPipeline method*), 1662
graph_feature_importance() (*evalml.pipelines.multiclass_classification_pipeline.MulticlassClassificationPipeline method*), 1578
graph_feature_importance() (*evalml.pipelines.MulticlassClassificationPipeline method*), 1730
graph_feature_importance() (*evalml.pipelines.pipeline_base.PipelineBase method*), 1585
graph_feature_importance() (*evalml.pipelines.PipelineBase method*), 1745
graph_feature_importance() (*evalml.pipelines.regression_pipeline.RegressionPipeline method*), 1593
graph_feature_importance() (*evalml.pipelines.RegressionPipeline method*), 1761
graph_feature_importance() (*evalml.pipelines.time_series_classification_pipelines.TimeSeries method*), 1600
graph_feature_importance() (*evalml.pipelines.time_series_classification_pipelines.TimeSeries method*), 1608
graph_feature_importance() (*evalml.pipelines.time_series_classification_pipelines.TimeSeries TimeSeriesBinaryClassificationPipeline*), 1608
graph_feature_importance() (*evalml.pipelines.time_series_classification_pipelines.TimeSeries TimeSeriesMulticlassClassificationPipeline*), 1615
graph_feature_importance() (*evalml.pipelines.time_series_pipeline_base.TimeSeriesPipelineB method*), 1624
graph_feature_importance() (*evalml.pipelines.time_series_regression_pipeline.TimeSeriesReg mesRegressionPipeline*), 1632
graph_feature_importance() (*evalml.pipelines.TimeSeriesBinaryClassificationPipeline method*), 1798
graph_feature_importance() (*evalml.pipelines.TimeSeriesClassificationPipeline method*), 1806
graph_feature_importance() (*evalml.pipelines.TimeSeriesMulticlassClassificationPipeline method*), 1819
graph_feature_importance() (*evalml.pipelines.TimeSeriesRegressionPipeline method*), 1828
graph_force_plot() (*in module evalml.model_understanding.force_plots*), 412
graph_partial_dependence() (*in module evalml.model_understanding*), 433
graph_partial_dependence() (*in module evalml.model_understanding.partial_dependence_functions*), 416
graph_permutation_importance() (*in module evalml.model_understanding*), 434
graph_permutation_importance() (*in module evalml.model_understanding.permutation_importance*), 420
graph_precision_recall_curve() (*in module evalml.model_understanding*), 434
graph_precision_recall_curve() (*in module evalml.model_understanding.metrics*), 414
graph_prediction_vs_actual() (*in module evalml.model_understanding*), 435
graph_prediction_vs_actual() (*in module evalml.model_understanding.visualizations*), 423
graph_prediction_vs_actual_over_time() (*in module evalml.model_understanding*), 435
graph_prediction_vs_actual_over_time() (*in module evalml.model_understanding.visualizations*),

424
graph_roc_curve() (*in module evalml.model_understanding*), 435
graph_roc_curve() (*in module evalml.model_understanding.metrics*), 414
graph_t_sne() (*in module evalml.model_understanding*), 436
graph_t_sne() (*in module evalml.model_understanding.visualizations*), 424
greater_is_better (*evalml.objectives.binary_classification_objective.BinaryClassificationObjective property*), 442
greater_is_better (*evalml.objectives.BinaryClassificationObjective property*), 551
greater_is_better (*evalml.objectives.multiclass_classification_objective.MulticlassClassificationObjective property*), 453
greater_is_better (*evalml.objectives.MulticlassClassificationObjective property*), 591
greater_is_better (*evalml.objectives.objective_base.ObjectiveBase property*), 455
greater_is_better (*evalml.objectives.ObjectiveBase property*), 593
greater_is_better (*evalml.objectives.regression_objective.RegressionObjective property*), 458
greater_is_better (*evalml.objectives.RegressionObjective property*), 612
greater_is_better (*evalml.objectives.time_series_regression_objective.TimeSeriesRegressionObjective property*), 529
GridSearchTuner (*class in evalml.tuners*), 1891
GridSearchTuner (*class in evalml.tuners.grid_search_tuner*), 1884

## H

handle_component_class() (*in module evalml.pipelines.components.utils*), 1330
handle_data_check_action_code() (*in module evalml.data_checks.utils*), 340
handle_dcao_parameter_type() (*evalml.data_checks.data_check_action_option.DCAOParameterType static method*), 294
handle_dcao_parameter_type() (*evalml.data_checks.DCAOParameterType static method*), 359
handle_float_categories_for_catboost() (*in module evalml.pipelines.components.utils*), 1330
handle_model_family() (*in module evalml.model_family*), 400
handle_model_family() (*in module evalml.model_family.utils*), 400
handle_problem_types() (*in module evalml.problem_types*), 1881
handle_problem_types() (*in module evalml.problem_types.utils*), 1877
has_dfs (*evalml.pipelines.component_graph.ComponentGraph property*), 1572
has_dfs (*evalml.pipelines.ComponentGraph property*), 1669

## I

IDColumnsDataCheck (*class in evalml.data_checks*), 361
IDColumnsDataCheck (*class in evalml.data_checks.id_columns_data_check*), 309
import_or_raise() (*in module evalml.utils*), 1912
import_or_raise() (*in module evalml.utils.gen_utils*), 1904
Imputer (*class in evalml.pipelines*), 1707
Imputer (*class in evalml.pipelines.components*), 1404
Imputer (*class in evalml.pipelines.components.transformers*), 1236
Imputer (*class in evalml.pipelines.components.transformers.imputers*), 1052
Imputer (*class in evalml.pipelines.components.transformers.imputers.impu*
1033
infer_feature_types() (*in module evalml.utils*), 1912
infer_feature_types() (*in module evalml.utils.woodwork_utils*), 1908
info() (*evalml.automl.engine.engine_base.JobLogger method*), 246
instantiate() (*evalml.pipelines.component_graph.ComponentGraph method*), 1572
instantiate() (*evalml.pipelines.ComponentGraph method*), 1669
InvalidTargetDataCheck (*class in evalml.data_checks*), 364
InvalidTargetDataCheck (*class in evalml.data_checks.invalid_target_data_check*), 312
inverse_transform() (*evalml.pipelines.binary_classification_pipeline.BinaryClassifica method*), 1558
inverse_transform() (*evalml.pipelines.BinaryClassificationPipeline method*), 1650
inverse_transform() (*evalml.pipelines.classification_pipeline.ClassificationPipeline method*), 1566
inverse_transform() (*evalml.pipelines.ClassificationPipeline method*), 1663
inverse_transform() (*evalml.pipelines.component_graph.ComponentGraph method*), 1572
inverse_transform() (*evalml.pipelines.ComponentGraph method*), 1669

inverse_transform() (*evalml.pipelines.components.LabelEncoder method*), 1412
inverse_transform() (*evalml.pipelines.components.LogTransformer method*), 1431
inverse_transform() (*evalml.pipelines.components.PolynomialDecomposer method*), 1454
inverse_transform() (*evalml.pipelines.components.STLDecomposer method*), 1503
inverse_transform() (*evalml.pipelines.components.transformers.encoders.label_encoder.LabelEncoder method*), 970
inverse_transform() (*evalml.pipelines.components.transformers.encoders.LabelEncoder method*), 986
inverse_transform() (*evalml.pipelines.components.transformers.LabelEncoder method*), 1240
inverse_transform() (*evalml.pipelines.components.transformers.LogTransformer method*), 1245
inverse_transform() (*evalml.pipelines.components.transformers.PolynomialDecomposer method*), 1269
inverse_transform() (*evalml.pipelines.components.transformers.preprocessing.decomposer method*), 1134
inverse_transform() (*evalml.pipelines.components.transformers.preprocessing.decomposer.base_decomposer method*), 1074
inverse_transform() (*evalml.pipelines.components.transformers.preprocessing.log_transformer method*), 1090
inverse_transform() (*evalml.pipelines.components.transformers.preprocessing.LogTransformer method*), 1149
inverse_transform() (*evalml.pipelines.components.transformers.preprocessing.polynomial_decomposer.PolynomialDecomposer method*), 1101
inverse_transform() (*evalml.pipelines.components.transformers.preprocessing.PolynomialDecomposer method*), 1158
inverse_transform() (*evalml.pipelines.components.transformers.preprocessing.stl_decomposer.STLDecomposer method*), 1110
inverse_transform() (*evalml.pipelines.components.transformers.preprocessing.STLDecomposer method*), 1166
inverse_transform() (*evalml.pipelines.components.transformers.STLDecomposer method*), 1298
inverse_transform() (*evalml.pipelines.multiclass_classification_pipeline.MulticlassCla method*), 1579
inverse_transform() (*evalml.pipelines.MulticlassClassificationPipeline method*), 1731
inverse_transform() (*evalml.pipelines.pipeline_base.PipelineBase method*), 1585
inverse_transform() (*evalml.pipelines.PipelineBase method*), 1745
inverse_transform() (*evalml.pipelines.regression_pipeline.RegressionPipeline method*), 1597
inverse_transform() (*evalml.pipelines.RegressionPipeline method*), 1761
inverse_transform() (*evalml.pipelines.time_series_classification_pipelines.TimeSeries method*), 1601
inverse_transform() (*evalml.pipelines.time_series_classification_pipelines.TimeSeries method*), 1608
inverse_transform() (*evalml.pipelines.time_series_classification_pipelines.TimeSeries method*), 1616
inverse_transform() (*evalml.pipelines.time_series_pipeline_base.TimeSeriesPipelineB method*), 1624
inverse_transform() (*evalml.pipelines.time_series_regression_pipeline.TimeSeriesReg method*), 1632
inverse_transform() (*evalml.pipelines.TimeSeriesBinaryClassificationPipeline method*), 1799
inverse_transform() (*evalml.pipelines.TimeSeriesClassificationPipeline method*), 1806
inverse_transform() (*evalml.pipelines.TimeSeriesMulticlassClassificationPipeline method*), 1819
inverse_transform() (*evalml.pipelines.TimeSeriesRegressionPipeline method*), 1828
is_all_numeric() (*in module evalml.utils*), 1912
is_all_numeric() (*in module evalml.utils.gen_utils*), 1904
is_binary() (*in module evalml.problem_types*), 1881
is_binary() (*in module evalml.problem_types.utils*), 1878
is_bounded_like_percentage (*evalml.objectives.binary_classification_objective.BinaryClassific property*), 442
is_bounded_like_percentage

(*evalml.objectives.BinaryClassificationObjective property*), 551
is_bounded_like_percentage (*evalml.objectives.multiclass_classification_objective.MulticlassClassificationObjective property*), 453
is_bounded_like_percentage (*evalml.objectives.MulticlassClassificationObjective property*), 591
is_bounded_like_percentage (*evalml.objectives.objective_base.ObjectiveBase property*), 455
is_bounded_like_percentage (*evalml.objectives.ObjectiveBase property*), 593
is_bounded_like_percentage (*evalml.objectives.regression_objective.RegressionObjective property*), 458
is_bounded_like_percentage (*evalml.objectives.RegressionObjective property*), 612
is_bounded_like_percentage (*evalml.objectives.time_series_regression_objective.TimeSeriesRegressionObjective property*), 529
is_cancelled (*evalml.automl.engine.cf_engine.CFComputation property*), 240
is_cancelled (*evalml.automl.engine.dask_engine.DaskComputation property*), 242
is_classification() (*in module evalml.problem_types*), 1881
is_classification() (*in module evalml.problem_types.utils*), 1878
is_closed (*evalml.automl.engine.cf_engine.CFClient property*), 239
is_closed (*evalml.automl.engine.cf_engine.CFEngine property*), 240
is_closed (*evalml.automl.engine.CFEngine property*), 251
is_closed (*evalml.automl.engine.dask_engine.DaskEngine property*), 242
is_closed (*evalml.automl.engine.DaskEngine property*), 252
is_cv (*evalml.preprocessing.data_splitters.KFold property*), 1860
is_cv (*evalml.preprocessing.data_splitters.no_split.NoSplit property*), 1853
is_cv (*evalml.preprocessing.data_splitters.NoSplit property*), 1861
is_cv (*evalml.preprocessing.data_splitters.sk_splitters.KFold property*), 1854
is_cv (*evalml.preprocessing.data_splitters.sk_splitters.StratifiedKFold property*), 1855
is_cv (*evalml.preprocessing.data_splitters.StratifiedKFold property*), 1862
is_cv (*evalml.preprocessing.data_splitters.time_series_split.TimeSeriesSplit property*), 1857
is_cv (*evalml.preprocessing.data_splitters.TimeSeriesSplit property*), 1864
is_cv (*evalml.preprocessing.data_splitters.training_validation_split.TrainingValidationSplit property*), 1859
is_cv (*evalml.preprocessing.data_splitters.TrainingValidationSplit property*), 1866
is_cv (*evalml.preprocessing.NoSplit property*), 1870
is_cv (*evalml.preprocessing.TimeSeriesSplit property*), 1873
is_cv (*evalml.preprocessing.TrainingValidationSplit property*), 1875
is_defined_for_problem_type() (*evalml.objectives.AccuracyBinary class method*), 535
is_defined_for_problem_type() (*evalml.objectives.AccuracyMulticlass class method*), 537
is_defined_for_problem_type() (*evalml.objectives.AUC class method*), 539
is_defined_for_problem_type() (*evalml.objectives.AUCMacro class method*), 541
is_defined_for_problem_type() (*evalml.objectives.AUCMicro class method*), 543
is_defined_for_problem_type() (*evalml.objectives.AUCWeighted class method*), 545
is_defined_for_problem_type() (*evalml.objectives.BalancedAccuracyBinary class method*), 547
is_defined_for_problem_type() (*evalml.objectives.BalancedAccuracyMulticlass class method*), 549
is_defined_for_problem_type() (*evalml.objectives.binary_classification_objective.BinaryClassificatio class method*), 442
is_defined_for_problem_type() (*evalml.objectives.BinaryClassificationObjective class method*), 551
is_defined_for_problem_type() (*evalml.objectives.cost_benefit_matrix.CostBenefitMatrix class method*), 445
is_defined_for_problem_type() (*evalml.objectives.CostBenefitMatrix class method*), 553
is_defined_for_problem_type() (*evalml.objectives.ExpVariance class method*), 555
is_defined_for_problem_type() (*evalml.objectives.F1 class method*), 557
is_defined_for_problem_type() (*evalml.objectives.F1Macro class method*),

559
is_defined_for_problem_type() (*evalml.objectives.F1Micro class method*), 561
is_defined_for_problem_type() (*evalml.objectives.F1Weighted class method*), 563
is_defined_for_problem_type() (*evalml.objectives.fraud_cost.FraudCost class method*), 448
is_defined_for_problem_type() (*evalml.objectives.FraudCost class method*), 565
is_defined_for_problem_type() (*evalml.objectives.Gini class method*), 569
is_defined_for_problem_type() (*evalml.objectives.lead_scoring.LeadScoring class method*), 450
is_defined_for_problem_type() (*evalml.objectives.LeadScoring class method*), 571
is_defined_for_problem_type() (*evalml.objectives.LogLossBinary class method*), 573
is_defined_for_problem_type() (*evalml.objectives.LogLossMulticlass class method*), 575
is_defined_for_problem_type() (*evalml.objectives.MAE class method*), 577
is_defined_for_problem_type() (*evalml.objectives.MAPE class method*), 579
is_defined_for_problem_type() (*evalml.objectives.MaxError class method*), 580
is_defined_for_problem_type() (*evalml.objectives.MCCBinary class method*), 582
is_defined_for_problem_type() (*evalml.objectives.MCCMulticlass class method*), 584
is_defined_for_problem_type() (*evalml.objectives.MeanSquaredLogError class method*), 586
is_defined_for_problem_type() (*evalml.objectives.MedianAE class method*), 588
is_defined_for_problem_type() (*evalml.objectives.MSE class method*), 589
is_defined_for_problem_type() (*evalml.objectives.multiclass_classification_objective.MulticlassClassificationObjective class method*), 453
is_defined_for_problem_type() (*evalml.objectives.MulticlassClassificationObjective class method*), 591
is_defined_for_problem_type() (*evalml.objectives.objective_base.ObjectiveBase class method*), 455
is_defined_for_problem_type() (*evalml.objectives.ObjectiveBase class method*), 593
is_defined_for_problem_type() (*evalml.objectives.Precision class method*), 596
is_defined_for_problem_type() (*evalml.objectives.PrecisionMacro class method*), 598
is_defined_for_problem_type() (*evalml.objectives.PrecisionMicro class method*), 599
is_defined_for_problem_type() (*evalml.objectives.PrecisionWeighted class method*), 601
is_defined_for_problem_type() (*evalml.objectives.R2 class method*), 603
is_defined_for_problem_type() (*evalml.objectives.Recall class method*), 605
is_defined_for_problem_type() (*evalml.objectives.RecallMacro class method*), 607
is_defined_for_problem_type() (*evalml.objectives.RecallMicro class method*), 608
is_defined_for_problem_type() (*evalml.objectives.RecallWeighted class method*), 610
is_defined_for_problem_type() (*evalml.objectives.regression_objective.RegressionObjective class method*), 458
is_defined_for_problem_type() (*evalml.objectives.RegressionObjective class method*), 612
is_defined_for_problem_type() (*evalml.objectives.RootMeanSquaredError class method*), 614
is_defined_for_problem_type() (*evalml.objectives.RootMeanSquaredLogError class method*), 615
is_defined_for_problem_type() (*evalml.objectives.sensitivity_low_alert.SensitivityLowAlert class method*), 461
is_defined_for_problem_type() (*evalml.objectives.SensitivityLowAlert class method*), 617
is_defined_for_problem_type() (*evalml.objectives.standard_metrics.AccuracyBinary class method*), 464

is_defined_for_problem_type() (*evalml.objectives.standard_metrics.AccuracyMulticlass class method*), 466
is_defined_for_problem_type() (*evalml.objectives.standard_metrics.AUC class method*), 468
is_defined_for_problem_type() (*evalml.objectives.standard_metrics.AUCMacro class method*), 470
is_defined_for_problem_type() (*evalml.objectives.standard_metrics.AUCMicro class method*), 472
is_defined_for_problem_type() (*evalml.objectives.standard_metrics.AUCWeighted class method*), 474
is_defined_for_problem_type() (*evalml.objectives.standard_metrics.BalancedAccuracyBinary class method*), 476
is_defined_for_problem_type() (*evalml.objectives.standard_metrics.BalancedAccuracyMulticlass class method*), 478
is_defined_for_problem_type() (*evalml.objectives.standard_metrics.ExpVariance class method*), 479
is_defined_for_problem_type() (*evalml.objectives.standard_metrics.F1 class method*), 481
is_defined_for_problem_type() (*evalml.objectives.standard_metrics.F1Macro class method*), 483
is_defined_for_problem_type() (*evalml.objectives.standard_metrics.F1Micro class method*), 485
is_defined_for_problem_type() (*evalml.objectives.standard_metrics.F1Weighted class method*), 487
is_defined_for_problem_type() (*evalml.objectives.standard_metrics.Gini class method*), 489
is_defined_for_problem_type() (*evalml.objectives.standard_metrics.LogLossBinary class method*), 491
is_defined_for_problem_type() (*evalml.objectives.standard_metrics.LogLossMulticlass class method*), 493
is_defined_for_problem_type() (*evalml.objectives.standard_metrics.MAE class method*), 494
is_defined_for_problem_type() (*evalml.objectives.standard_metrics.MAPE class method*), 496
is_defined_for_problem_type() (*evalml.objectives.standard_metrics.MaxError class method*), 498
is_defined_for_problem_type() (*evalml.objectives.standard_metrics.MCCBinary class method*), 500
is_defined_for_problem_type() (*evalml.objectives.standard_metrics.MCCMulticlass class method*), 502
is_defined_for_problem_type() (*evalml.objectives.standard_metrics.MeanSquaredLogError class method*), 504
is_defined_for_problem_type() (*evalml.objectives.standard_metrics.MedianAE class method*), 505
is_defined_for_problem_type() (*evalml.objectives.standard_metrics.MSE class method*), 507
is_defined_for_problem_type() (*evalml.objectives.standard_metrics.Precision class method*), 509
is_defined_for_problem_type() (*evalml.objectives.standard_metrics.PrecisionMacro class method*), 511
is_defined_for_problem_type() (*evalml.objectives.standard_metrics.PrecisionMicro class method*), 513
is_defined_for_problem_type() (*evalml.objectives.standard_metrics.PrecisionWeighted class method*), 514
is_defined_for_problem_type() (*evalml.objectives.standard_metrics.R2 class method*), 516
is_defined_for_problem_type() (*evalml.objectives.standard_metrics.Recall class method*), 518
is_defined_for_problem_type() (*evalml.objectives.standard_metrics.RecallMacro class method*), 520
is_defined_for_problem_type() (*evalml.objectives.standard_metrics.RecallMicro class method*), 522
is_defined_for_problem_type() (*evalml.objectives.standard_metrics.RecallWeighted class method*), 523
is_defined_for_problem_type() (*evalml.objectives.standard_metrics.RootMeanSquaredError class method*), 525
is_defined_for_problem_type() (*evalml.objectives.standard_metrics.RootMeanSquaredLogError class method*), 527
is_defined_for_problem_type() (*evalml.objectives.time_series_regression_objective.TimeSeriesRe class method*), 529
is_freq_valid() (*evalml.pipelines.components.PolynomialDecomposer class method*), 1454
is_freq_valid() (*evalml.pipelines.components.STLDecomposer*

class method), 1503
is_freq_valid() (evalml.pipelines.components.transformers.PolynomialDecomposer class method), 1269
is_freq_valid() (evalml.pipelines.components.transformers.preprocessing.DateTimeFeaturizer class method), 1134
is_freq_valid() (evalml.pipelines.components.transformers.preprocessing.DelayedFeatureTransformer class method), 1074
is_freq_valid() (evalml.pipelines.components.transformers.preprocessing.PolynomialDecomposer class method), 1101
is_freq_valid() (evalml.pipelines.components.transformers.preprocessing.RandomOversampler class method), 1159
is_freq_valid() (evalml.pipelines.components.transformers.preprocessing.TimeSeriesRegularizer class method), 1110
is_freq_valid() (evalml.pipelines.components.transformers.preprocessing.STLDecomposer class method), 1166
is_freq_valid() (evalml.pipelines.components.transformers.STLDecomposer class method), 1298
is_multiclass() (in module evalml.problem_types), 1881
is_multiclass() (in module evalml.problem_types.utils), 1878
is_regression() (in module evalml.problem_types), 1882
is_regression() (in module evalml.problem_types.utils), 1879
is_search_space_exhausted() (evalml.tuners.grid_search_tuner.GridSearchTuner method), 1885
is_search_space_exhausted() (evalml.tuners.GridSearchTuner method), 1893
is_search_space_exhausted() (evalml.tuners.random_search_tuner.RandomSearchTuner method), 1887
is_search_space_exhausted() (evalml.tuners.RandomSearchTuner method), 1894
is_search_space_exhausted() (evalml.tuners.skopt_tuner.SKOptTuner method), 1889
is_search_space_exhausted() (evalml.tuners.SKOptTuner method), 1896
is_search_space_exhausted() (evalml.tuners.Tuner method), 1897
is_search_space_exhausted() (evalml.tuners.tuner.Tuner method), 1890
is_time_series() (in module evalml.problem_types), 1882
is_time_series() (in module evalml.problem_types.utils), 1879
is_tree_estimator() (evalml.model_family.model_family.ModelFamily method), 399
is_tree_estimator() (evalml.model_family.ModelFamily method), 401
IterativeAlgorithm (class in evalml.automl.automl_algorithm), 236
IterativeAlgorithm (class in evalml.automl.automl_algorithm.iterative_algorithm), 228


## J


JobLogger (class in evalml.automl.engine.engine_base), 246
jupyter_check() (in module evalml.utils), 1912
jupyter_check() (in module evalml.utils.gen_utils), 1905


## K


KFold (class in evalml.preprocessing.data_splitters), 1860
KFold (class in evalml.preprocessing.data_splitters.sk_splitters), 1854
KNeighborsClassifier (class in evalml.pipelines), 1709
KNeighborsClassifier (class in evalml.pipelines.components), 1407
KNeighborsClassifier (class in evalml.pipelines.components.estimators), 904
KNeighborsClassifier (class in evalml.pipelines.components.estimators.classifiers), 713
KNeighborsClassifier (class in evalml.pipelines.components.estimators.classifiers.kneighbors_cl 663
KNNImputer (class in evalml.pipelines.components.transformers.imputers), 1055
KNNImputer (class in evalml.pipelines.components.transformers.imputers.k 1036


## L


LabelEncoder (class in evalml.pipelines.components), 1410
LabelEncoder (class in evalml.pipelines.components.transformers), 1238
LabelEncoder (class in evalml.pipelines.components.transformers.encoders), 984
LabelEncoder (class in evalml.pipelines.components.transformers.encoders.label_encode 968
last_component_input_logical_types (evalml.pipelines.component_graph.ComponentGraph property), 1573

last_component_input_logical_types (*evalml.pipelines.ComponentGraph property*), 1670
LeadScoring (*class in evalml.objectives*), 570
LeadScoring (*class in evalml.objectives.lead_scoring*), 449
LightGBMClassifier (*class in evalml.pipelines*), 1713
LightGBMClassifier (*class in evalml.pipelines.components*), 1413
LightGBMClassifier (*class in evalml.pipelines.components.estimators*), 907
LightGBMClassifier (*class in evalml.pipelines.components.estimators.classifiers*), 717
LightGBMClassifier (*class in evalml.pipelines.components.estimators.classifiers.lightgbm_classifier*), 667
LightGBMRegressor (*class in evalml.pipelines*), 1716
LightGBMRegressor (*class in evalml.pipelines.components*), 1416
LightGBMRegressor (*class in evalml.pipelines.components.estimators*), 911
LightGBMRegressor (*class in evalml.pipelines.components.estimators.regressors*), 826
LightGBMRegressor (*class in evalml.pipelines.components.estimators.regressors.lightgbm_regressor*), 770
LinearDiscriminantAnalysis (*class in evalml.pipelines.components*), 1420
LinearDiscriminantAnalysis (*class in evalml.pipelines.components.transformers*), 1241
LinearDiscriminantAnalysis (*class in evalml.pipelines.components.transformers.dimensionality_reduction*), 963
LinearDiscriminantAnalysis (*class in evalml.pipelines.components.transformers.dimensionality_reduction.linear_discriminant_analysis*), 957
LinearRegressor (*class in evalml.pipelines*), 1720
LinearRegressor (*class in evalml.pipelines.components*), 1422
LinearRegressor (*class in evalml.pipelines.components.estimators*), 915
LinearRegressor (*class in evalml.pipelines.components.estimators.regressors*), 830
LinearRegressor (*class in evalml.pipelines.components.estimators.regressors.linear_regressor*), 774
load() (*evalml.automl.automl_search.AutoMLSearch static method*), 262
load() (*evalml.automl.AutoMLSearch static method*), 278
load() (*evalml.AutoMLSearch static method*), 1918
load() (*evalml.pipelines.ARIMARegressor static method*), 1644
load() (*evalml.pipelines.binary_classification_pipeline.BinaryClassificati static method*), 1558
load() (*evalml.pipelines.BinaryClassificationPipeline static method*), 1650
load() (*evalml.pipelines.CatBoostClassifier static method*), 1654
load() (*evalml.pipelines.CatBoostRegressor static method*), 1658
load() (*evalml.pipelines.classification_pipeline.ClassificationPipeline static method*), 1566
load() (*evalml.pipelines.ClassificationPipeline static method*), 1663
load() (*evalml.pipelines.components.ARIMARegressor static method*), 1339
load() (*evalml.pipelines.components.BaselineClassifier static method*), 1343
load() (*evalml.pipelines.components.BaselineRegressor static method*), 1345
load() (*evalml.pipelines.components.CatBoostClassifier static method*), 1349
load() (*evalml.pipelines.components.CatBoostRegressor static method*), 1352
load() (*evalml.pipelines.components.component_base.ComponentBase static method*), 1325
load() (*evalml.pipelines.components.ComponentBase static method*), 1355
load() (*evalml.pipelines.components.DateTimeFeaturizer static method*), 1358
load() (*evalml.pipelines.components.DecisionTreeClassifier static method*), 1361
load() (*evalml.pipelines.components.DecisionTreeRegressor static method*), 1365
load() (*evalml.pipelines.components.DFSTransformer static method*), 1368
load() (*evalml.pipelines.components.DropColumns static method*), 1371
load() (*evalml.pipelines.components.DropNaNRowsTransformer static method*), 1373
load() (*evalml.pipelines.components.DropNullColumns static method*), 1376
load() (*evalml.pipelines.components.DropRowsTransformer static method*), 1378
load() (*evalml.pipelines.components.ElasticNetClassifier static method*), 1381
load() (*evalml.pipelines.components.ElasticNetRegressor static method*), 1384
load() (*evalml.pipelines.components.EmailFeaturizer static method*), 1387

load() (evalml.pipelines.components.ensemble.stacked_ensemble_base.StackedEnsembleBase static method), 621
load() (evalml.pipelines.components.ensemble.stacked_ensemble_classifier.StackedEnsembleClassifier static method), 625
load() (evalml.pipelines.components.ensemble.stacked_ensemble_regressor.StackedEnsembleRegressor static method), 630
load() (evalml.pipelines.components.ensemble.StackedEnsembleBase static method), 633
load() (evalml.pipelines.components.ensemble.StackedEnsembleClassifier static method), 637
load() (evalml.pipelines.components.ensemble.StackedEnsembleRegressor static method), 641
load() (evalml.pipelines.components.Estimator static method), 1390
load() (evalml.pipelines.components.estimators.ARIMARegressor static method), 861
load() (evalml.pipelines.components.estimators.BaselineClassifier static method), 865
load() (evalml.pipelines.components.estimators.BaselineRegressor static method), 868
load() (evalml.pipelines.components.estimators.CatBoostClassifier static method), 871
load() (evalml.pipelines.components.estimators.CatBoostRegressor static method), 874
load() (evalml.pipelines.components.estimators.classifiers.BaselineClassifier static method), 645
load() (evalml.pipelines.components.estimators.classifiers.BinaryClassifier static method), 698
load() (evalml.pipelines.components.estimators.classifiers.CatBoostClassifier static method), 649
load() (evalml.pipelines.components.estimators.classifiers.CatBoostClassifier static method), 701
load() (evalml.pipelines.components.estimators.classifiers.DecisionTreeClassifier static method), 653
load() (evalml.pipelines.components.estimators.classifiers.DecisionTreeClassifier static method), 705
load() (evalml.pipelines.components.estimators.classifiers.ElasticNetClassifier static method), 657
load() (evalml.pipelines.components.estimators.classifiers.ElasticNetClassifier static method), 709
load() (evalml.pipelines.components.estimators.classifiers.ExtraTreesClassifier static method), 661
load() (evalml.pipelines.components.estimators.classifiers.ExtraTreesClassifier static method), 712
load() (evalml.pipelines.components.estimators.classifiers.KNeighborsClassifier static method), 665
load() (evalml.pipelines.components.estimators.classifiers.KNeighborsClassifier static method), 716
load() (evalml.pipelines.components.estimators.classifiers.LightGBMClassifier static method), 670
load() (evalml.pipelines.components.estimators.classifiers.LightGBMClassifier static method), 720
load() (evalml.pipelines.components.estimators.classifiers.LogisticRegressionClassifier static method), 673
load() (evalml.pipelines.components.estimators.classifiers.LogisticRegressionClassifier static method), 723
load() (evalml.pipelines.components.estimators.classifiers.RandomForestClassifier static method), 726
load() (evalml.pipelines.components.estimators.classifiers.rf_classifier.RandomForestClassifier static method), 677
load() (evalml.pipelines.components.estimators.classifiers.svm_classifier.SVMClassifier static method), 681
load() (evalml.pipelines.components.estimators.classifiers.SVMClassifier static method), 730
load() (evalml.pipelines.components.estimators.classifiers.vowpal_wabbit_base_classifier.VowpalWabbitBaseClassifier static method), 684
load() (evalml.pipelines.components.estimators.classifiers.vowpal_wabbit_base_classifier.VowpalWabbitBaseClassifier static method), 687
load() (evalml.pipelines.components.estimators.classifiers.vowpal_wabbit_base_classifier.VowpalWabbitBaseClassifier static method), 691
load() (evalml.pipelines.components.estimators.classifiers.VowpalWabbitBaseClassifier static method), 733
load() (evalml.pipelines.components.estimators.classifiers.VowpalWabbitMulticlassClassifier static method), 736
load() (evalml.pipelines.components.estimators.classifiers.xgboost_classifier.XGBoostClassifier static method), 695
load() (evalml.pipelines.components.estimators.classifiers.XGBoostClassifier static method), 739
load() (evalml.pipelines.components.estimators.DecisionTreeClassifier static method), 878
load() (evalml.pipelines.components.estimators.DecisionTreeRegressor static method), 882
load() (evalml.pipelines.components.estimators.ElasticNetClassifier static method), 885
load() (evalml.pipelines.components.estimators.ElasticNetRegressor static method), 888
load() (evalml.pipelines.components.estimators.Estimator static method), 892
load() (evalml.pipelines.components.estimators.estimator.Estimator static method), 856
load() (evalml.pipelines.components.estimators.ExponentialSmoothingRegressor static method), 895
load() (evalml.pipelines.components.estimators.ExtraTreesClassifier static method), 899
load() (evalml.pipelines.components.estimators.ExtraTreesRegressor static method), 903
load() (evalml.pipelines.components.estimators.KNeighborsClassifier static method), 906
load() (evalml.pipelines.components.estimators.LightGBMClassifier static method), 910
load() (evalml.pipelines.components.estimators.LightGBMRegressor static method), 914
load() (evalml.pipelines.components.estimators.LinearRegressor static method), 917
load() (evalml.pipelines.components.estimators.LogisticRegressionClassifier static method), 920
load() (evalml.pipelines.components.estimators.LogisticRegressionClassifier static method), 924

load() (*evalml.pipelines.components.estimators.RandomForestClassifier static method*), 927
load() (*evalml.pipelines.components.estimators.RandomForestRegressor static method*), 930
load() (*evalml.pipelines.components.estimators.regressors.arima_regressor.ARIMARegressor static method*), 745
load() (*evalml.pipelines.components.estimators.regressors.arima_regressor.AutoARIMARegressor static method*), 804
load() (*evalml.pipelines.components.estimators.regressors.baseline_regressor.BaselineRegressor static method*), 748
load() (*evalml.pipelines.components.estimators.regressors.catboost_regressor.CatBoostRegressor static method*), 807
load() (*evalml.pipelines.components.estimators.regressors.exponential_smoothing_regressor.ExponentialSmoothingRegressor static method*), 752
load() (*evalml.pipelines.components.estimators.regressors.extra_trees_regressor.ExtraTreesRegressor static method*), 811
load() (*evalml.pipelines.components.estimators.regressors.decision_tree_regressor.DecisionTreeRegressor static method*), 756
load() (*evalml.pipelines.components.estimators.regressors.linear_regressor.LinearRegressor static method*), 815
load() (*evalml.pipelines.components.estimators.regressors.elastic_net_regressor.ElasticNetRegressor static method*), 760
load() (*evalml.pipelines.components.estimators.regressors.lightgbm_regressor.LightGBMRegressor static method*), 818
load() (*evalml.pipelines.components.estimators.regressors.exponential_smoothing_regressor.ExponentialSmoothingRegressor static method*), 765
load() (*evalml.pipelines.components.estimators.regressors.extra_trees_regressor.ExtraTreesRegressor static method*), 769
load() (*evalml.pipelines.components.estimators.regressors.decision_tree_regressor.DecisionTreeRegressor static method*), 822
load() (*evalml.pipelines.components.estimators.regressors.linear_regressor.LinearRegressor static method*), 825
load() (*evalml.pipelines.components.estimators.regressors.lightgbm_regressor.LightGBMRegressor static method*), 773
load() (*evalml.pipelines.components.estimators.regressors.catboost_regressor.CatBoostRegressor static method*), 829
load() (*evalml.pipelines.components.estimators.regressors.elastic_net_regressor.ElasticNetRegressor static method*), 776
load() (*evalml.pipelines.components.estimators.regressors.svm_regressor.SVMRegressor static method*), 832
load() (*evalml.pipelines.components.estimators.regressors.prophet_regressor.ProphetRegressor static method*), 781
load() (*evalml.pipelines.components.estimators.regressors.xgboost_regressor.XGBoostRegressor static method*), 836
load() (*evalml.pipelines.components.estimators.regressors.vowpal_wabbit_regressor.VowpalWabbitRegressor static method*), 839
load() (*evalml.pipelines.components.estimators.regressors.time_series_baseline_regressor.TimeSeriesBaselineRegressor static method*), 785
load() (*evalml.pipelines.components.estimators.regressors.time_series_regressors.TimeSeriesRegressor static method*), 788
load() (*evalml.pipelines.components.estimators.regressors.baseline_regressor.TimeSeriesBaselineEstimator static method*), 842
load() (*evalml.pipelines.components.estimators.regressors.time_series_regressors.TimeSeriesRegressor static method*), 792
load() (*evalml.pipelines.components.estimators.regressors.vowpal_wabbit_regressors.VowpalWabbitRegressor static method*), 795
load() (*evalml.pipelines.components.estimators.regressors.VowpalWabbitRegressor static method*), 849
load() (*evalml.pipelines.components.estimators.regressors.xgboost_regressor.XGBoostRegressor static method*), 799
load() (*evalml.pipelines.components.estimators.regressors.XGBoostRegressor static method*), 852
load() (*evalml.pipelines.components.SVMClassifier static method*), 933
load() (*evalml.pipelines.components.SVMRegressor static method*), 936
load() (*evalml.pipelines.components.TimeSeriesBaselineEstimator static method*), 940
load() (*evalml.pipelines.components.VowpalWabbitBinaryClassifier static method*), 943
load() (*evalml.pipelines.components.VowpalWabbitMulticlassClassifier static method*), 946
load() (*evalml.pipelines.components.VowpalWabbitRegressor static method*), 949
load() (*evalml.pipelines.components.XGBoostClassifier static method*), 952
load() (*evalml.pipelines.components.XGBoostRegressor static method*), 955
load() (*evalml.pipelines.components.ExponentialSmoothingRegressor static method*), 1393
load() (*evalml.pipelines.components.ExtraTreesClassifier static method*), 1397
load() (*evalml.pipelines.components.ExtraTreesRegressor static method*), 1401
load() (*evalml.pipelines.components.FeatureSelector static method*), 1403
load() (*evalml.pipelines.components.Imputer static method*), 1406
load() (*evalml.pipelines.components.KNeighborsClassifier static method*), 1409
load() (*evalml.pipelines.components.LabelEncoder static method*), 1412
load() (*evalml.pipelines.components.LightGBMClassifier static method*), 1415
load() (*evalml.pipelines.components.LightGBMRegressor static method*), 1419
load() (*evalml.pipelines.components.LinearDiscriminantAnalysis static method*), 1422
load() (*evalml.pipelines.components.LinearRegressor static method*), 1424
load() (*evalml.pipelines.components.LogisticRegressionClassifier static method*), 1428
load() (*evalml.pipelines.components.LogTransformer static method*), 1431
load() (*evalml.pipelines.components.TimeSeriesBaselineEstimator static method*), 1433

load() (*evalml.pipelines.components.NaturalLanguageFeaturizer static method*), 1436
load() (*evalml.pipelines.components.OneHotEncoder static method*), 1439
load() (*evalml.pipelines.components.OrdinalEncoder static method*), 1442
load() (*evalml.pipelines.components.Oversampler static method*), 1445
load() (*evalml.pipelines.components.PCA static method*), 1447
load() (*evalml.pipelines.components.PerColumnImputer static method*), 1450
load() (*evalml.pipelines.components.PolynomialDecomposer static method*), 1454
load() (*evalml.pipelines.components.ProphetRegressor static method*), 1458
load() (*evalml.pipelines.components.RandomForestClassifier static method*), 1461
load() (*evalml.pipelines.components.RandomForestRegressor static method*), 1464
load() (*evalml.pipelines.components.ReplaceNullableTypes static method*), 1467
load() (*evalml.pipelines.components.RFClassifierRFESelector static method*), 1470
load() (*evalml.pipelines.components.RFClassifierSelectFromModel static method*), 1473
load() (*evalml.pipelines.components.RFRegressorRFESelector static method*), 1476
load() (*evalml.pipelines.components.RFRegressorSelectFromModel static method*), 1479
load() (*evalml.pipelines.components.SelectByType static method*), 1481
load() (*evalml.pipelines.components.SelectColumns static method*), 1484
load() (*evalml.pipelines.components.SimpleImputer static method*), 1486
load() (*evalml.pipelines.components.StackedEnsembleBase static method*), 1489
load() (*evalml.pipelines.components.StackedEnsembleClassifier static method*), 1493
load() (*evalml.pipelines.components.StackedEnsembleRegressor static method*), 1497
load() (*evalml.pipelines.components.StandardScaler static method*), 1499
load() (*evalml.pipelines.components.STLDecomposer static method*), 1504
load() (*evalml.pipelines.components.SVMClassifier static method*), 1507
load() (*evalml.pipelines.components.SVMRegressor static method*), 1510
load() (*evalml.pipelines.components.TargetEncoder static method*), 1513
load() (*evalml.pipelines.components.TargetImputer static method*), 1516
load() (*evalml.pipelines.components.TimeSeriesBaselineEstimator static method*), 1519
load() (*evalml.pipelines.components.TimeSeriesFeaturizer static method*), 1522
load() (*evalml.pipelines.components.TimeSeriesImputer static method*), 1525
load() (*evalml.pipelines.components.TimeSeriesRegularizer static method*), 1528
load() (*evalml.pipelines.components.Transformer static method*), 1531
load() (*evalml.pipelines.components.transformers.column_selectors.Colu static method*), 1203
load() (*evalml.pipelines.components.transformers.column_selectors.Drop static method*), 1205
load() (*evalml.pipelines.components.transformers.column_selectors.Selec static method*), 1208
load() (*evalml.pipelines.components.transformers.column_selectors.Selec static method*), 1210
load() (*evalml.pipelines.components.transformers.DateTimeFeaturizer static method*), 1217
load() (*evalml.pipelines.components.transformers.DFSTransformer static method*), 1220
load() (*evalml.pipelines.components.transformers.dimensionality_reductio static method*), 959
load() (*evalml.pipelines.components.transformers.dimensionality_reductio static method*), 965
load() (*evalml.pipelines.components.transformers.dimensionality_reductio static method*), 967
load() (*evalml.pipelines.components.transformers.dimensionality_reductio static method*), 962
load() (*evalml.pipelines.components.transformers.DropColumns static method*), 1223
load() (*evalml.pipelines.components.transformers.DropNaNRowsTransfor static method*), 1225
load() (*evalml.pipelines.components.transformers.DropNullColumns static method*), 1227
load() (*evalml.pipelines.components.transformers.DropRowsTransformer static method*), 1230
load() (*evalml.pipelines.components.transformers.EmailFeaturizer static method*), 1232
load() (*evalml.pipelines.components.transformers.encoders.label_encoder static method*), 970
load() (*evalml.pipelines.components.transformers.encoders.LabelEncoder static method*), 986
load() (*evalml.pipelines.components.transformers.encoders.onehot_encod static method*), 974
load() (*evalml.pipelines.components.transformers.encoders.OneHotEncod static method*), 989
load() (*evalml.pipelines.components.transformers.encoders.ordinal_encod static method*), 979
load() (*evalml.pipelines.components.transformers.encoders.OrdinalEncod static method*), 993
load() (*evalml.pipelines.components.transformers.encoders.target_encode static method*), 983

load() (evalml.pipelines.components.transformers.encoders.TargetEncoder static method), 995
load() (evalml.pipelines.components.transformers.feature_selection.FeatureSelector static method), 998
load() (evalml.pipelines.components.transformers.feature_selection.FeatureSelection static method), 1018
load() (evalml.pipelines.components.transformers.feature_selection.RecursiveFeatureElimination static method), 1002
load() (evalml.pipelines.components.transformers.feature_selection.RFCategoricalEncoderSelector static method), 1005
load() (evalml.pipelines.components.transformers.feature_selection.RFNumericalEncoderSelector static method), 1008
load() (evalml.pipelines.components.transformers.feature_selection.RFClassifierSelectFromModel static method), 1012
load() (evalml.pipelines.components.transformers.feature_selection.RFRegressorSelectFromModel static method), 1015
load() (evalml.pipelines.components.transformers.feature_selection.RFClassifierRFESelector static method), 1022
load() (evalml.pipelines.components.transformers.feature_selection.RFClassifierSelectFromModel static method), 1025
load() (evalml.pipelines.components.transformers.feature_selection.RFRegressorRFESelector static method), 1028
load() (evalml.pipelines.components.transformers.feature_selection.RFRegressorSelectFromModel static method), 1031
load() (evalml.pipelines.components.transformers.FeatureSelector static method), 1235
load() (evalml.pipelines.components.transformers.Imputer static method), 1238
load() (evalml.pipelines.components.transformers.imputers.Imputer static method), 1054
load() (evalml.pipelines.components.transformers.imputers.SimpleImputer static method), 1035
load() (evalml.pipelines.components.transformers.imputers.KNNImputer static method), 1038
load() (evalml.pipelines.components.transformers.imputers.KNNImputer static method), 1056
load() (evalml.pipelines.components.transformers.imputers.PerColumnImputer static method), 1041
load() (evalml.pipelines.components.transformers.imputers.PerColumnImputer static method), 1059
load() (evalml.pipelines.components.transformers.imputers.SimpleImputer static method), 1044
load() (evalml.pipelines.components.transformers.imputers.TimeSeriesImputer static method), 1061
load() (evalml.pipelines.components.transformers.imputers.TimeSeriesImputer static method), 1047
load() (evalml.pipelines.components.transformers.imputers.TimeSeriesImputer static method), 1064
load() (evalml.pipelines.components.transformers.imputers.TimeSeriesImputer static method), 1051
load() (evalml.pipelines.components.transformers.imputers.TimeSeriesImputer static method), 1067
load() (evalml.pipelines.components.transformers.LabelEncoder static method), 1240
load() (evalml.pipelines.components.transformers.LinearDiscriminantAna static method), 1243
load() (evalml.pipelines.components.transformers.LogTransformer static method), 1245
load() (evalml.pipelines.components.transformers.LSA static method), 1248
load() (evalml.pipelines.components.transformers.NaturalLanguageFeatu static method), 1250
load() (evalml.pipelines.components.transformers.OneHotEncoder static method), 1253
load() (evalml.pipelines.components.transformers.OrdinalEncoder static method), 1257
load() (evalml.pipelines.components.transformers.RFClassifierSelectFromModel static method), 1260
load() (evalml.pipelines.components.transformers.RFRegressorSelectFromModel static method), 1262
load() (evalml.pipelines.components.transformers.PerColumnImputer static method), 1265
load() (evalml.pipelines.components.transformers.PolynomialDecomposer static method), 1269
load() (evalml.pipelines.components.transformers.preprocessing.datetime_ static method), 1070
load() (evalml.pipelines.components.transformers.preprocessing.DateTim static method), 1130
load() (evalml.pipelines.components.transformers.preprocessing.Decomp static method), 1134
load() (evalml.pipelines.components.transformers.preprocessing.decompo static method), 1075
load() (evalml.pipelines.components.transformers.preprocessing.DFSTran static method), 1137
load() (evalml.pipelines.components.transformers.preprocessing.drop_na static method), 1078
load() (evalml.pipelines.components.transformers.preprocessing.drop_nul static method), 1081
load() (evalml.pipelines.components.transformers.preprocessing.drop_row static method), 1084
load() (evalml.pipelines.components.transformers.preprocessing.DropNaN static method), 1140
load() (evalml.pipelines.components.transformers.preprocessing.DropNul static method), 1142
load() (evalml.pipelines.components.transformers.preprocessing.DropRow static method), 1145
load() (evalml.pipelines.components.transformers.preprocessing.EmailFea static method), 1147
load() (evalml.pipelines.components.transformers.preprocessing.featureto static method), 1087
load() (evalml.pipelines.components.transformers.preprocessing.log_trans static method), 1090
load() (evalml.pipelines.components.transformers.preprocessing.LogTrans static method), 1149
load() (evalml.pipelines.components.transformers.preprocessing.LSA static method), 1152
load() (evalml.pipelines.components.transformers.preprocessing.lsa.LSA static method), 1093

load() (*evalml.pipelines.components.transformers.preprocessing.NaturalLanguageFeaturizer static method*), 1096
load() (*evalml.pipelines.components.transformers.preprocessing.NaturalLanguageFeaturizer static method*), 1154
load() (*evalml.pipelines.components.transformers.preprocessing.PolynomialDecomposer static method*), 1101
load() (*evalml.pipelines.components.transformers.preprocessing.PolynomialDecomposer static method*), 1159
load() (*evalml.pipelines.components.transformers.preprocessing.RepeatNullInstances static method*), 1105
load() (*evalml.pipelines.components.transformers.preprocessing.RepeatNullInstances static method*), 1162
load() (*evalml.pipelines.components.transformers.preprocessing.STLDecomposer static method*), 1111
load() (*evalml.pipelines.components.transformers.preprocessing.STLDecomposer static method*), 1167
load() (*evalml.pipelines.components.transformers.preprocessing.TextTokenizer static method*), 1114
load() (*evalml.pipelines.components.transformers.preprocessing.TextTokenizer static method*), 1170
load() (*evalml.pipelines.components.transformers.preprocessing.TimeSeriesFeaturizer static method*), 1118
load() (*evalml.pipelines.components.transformers.preprocessing.TimeSeriesFeaturizer static method*), 1121
load() (*evalml.pipelines.components.transformers.preprocessing.TimeSeriesFeaturizer static method*), 1173
load() (*evalml.pipelines.components.transformers.preprocessing.TimeSeriesRegularizer static method*), 1176
load() (*evalml.pipelines.components.transformers.preprocessing.transform_primitive_components.EmailFeaturizer static method*), 1124
load() (*evalml.pipelines.components.transformers.preprocessing.transform_primitive_components.URLFeaturizer static method*), 1127
load() (*evalml.pipelines.components.transformers.preprocessing.URLFeaturizer static method*), 1179
load() (*evalml.pipelines.components.transformers.ReplaceNullableTypes static method*), 1272
load() (*evalml.pipelines.components.transformers.RFClassifierRFSelector static method*), 1275
load() (*evalml.pipelines.components.transformers.RFClassifierSelectFromModel static method*), 1278
load() (*evalml.pipelines.components.transformers.RFRegressorRFSelector static method*), 1281
load() (*evalml.pipelines.components.transformers.RFRegressorSelectFromModel static method*), 1284
load() (*evalml.pipelines.components.transformers.samplers.base_sampler.BaseSampler static method*), 1182
load() (*evalml.pipelines.components.transformers.samplers.base_sampler.BaseSampler static method*), 1191
load() (*evalml.pipelines.components.transformers.samplers.base_sampler.BaseOversampler static method*), 1185
load() (*evalml.pipelines.components.transformers.samplers.base_sampler.BaseOversampler static method*), 1194
load() (*evalml.pipelines.components.transformers.samplers.base_sampler.BaseOversampler static method*), 1188
load() (*evalml.pipelines.components.transformers.scalers.StandardScaler static method*), 1197
load() (*evalml.pipelines.components.transformers.scalers.StandardScaler static method*), 1200
load() (*evalml.pipelines.components.transformers.SelectByType static method*), 1287
load() (*evalml.pipelines.components.transformers.SelectColumns static method*), 1289
load() (*evalml.pipelines.components.transformers.SimpleImputer static method*), 1291
load() (*evalml.pipelines.components.transformers.StandardScaler static method*), 1294
load() (*evalml.pipelines.components.transformers.STLDecomposer static method*), 1299
load() (*evalml.pipelines.components.transformers.TargetEncoder static method*), 1302
load() (*evalml.pipelines.components.transformers.TargetImputer static method*), 1305
load() (*evalml.pipelines.components.transformers.TimeSeriesFeaturizer static method*), 1308
load() (*evalml.pipelines.components.transformers.TimeSeriesImputer static method*), 1311
load() (*evalml.pipelines.components.transformers.TimeSeriesRegularizer static method*), 1314
load() (*evalml.pipelines.components.transformers.Transformer static method*), 1317
load() (*evalml.pipelines.components.transformers.transformer.Transformer static method*), 1213
load() (*evalml.pipelines.components.Undersampler static method*), 1320
load() (*evalml.pipelines.components.URLFeaturizer static method*), 1322
load() (*evalml.pipelines.components.Undersampler static method*), 1534
load() (*evalml.pipelines.components.URLFeaturizer static method*), 1536
load() (*evalml.pipelines.components.VowpalWabbitBinaryClassifier static method*), 1539
load() (*evalml.pipelines.components.VowpalWabbitMulticlassClassifier static method*), 1542
load() (*evalml.pipelines.components.VowpalWabbitRegressor static method*), 1545
load() (*evalml.pipelines.components.XGBoostClassifier static method*), 1548
load() (*evalml.pipelines.components.XGBoostRegressor static method*), 1551
load() (*evalml.pipelines.DecisionTreeClassifier static method*), 1673
load() (*evalml.pipelines.DecisionTreeRegressor static method*), 1677
load() (*evalml.pipelines.DFSTransformer static method*), 1680
load() (*evalml.pipelines.DropNaNRowsTransformer static method*), 1683

load() (*evalml.pipelines.ElasticNetClassifier static method*), 1686
load() (*evalml.pipelines.ElasticNetRegressor static method*), 1689
load() (*evalml.pipelines.Estimator static method*), 1692
load() (*evalml.pipelines.ExponentialSmoothingRegressor static method*), 1695
load() (*evalml.pipelines.ExtraTreesClassifier static method*), 1699
load() (*evalml.pipelines.ExtraTreesRegressor static method*), 1703
load() (*evalml.pipelines.FeatureSelector static method*), 1706
load() (*evalml.pipelines.Imputer static method*), 1709
load() (*evalml.pipelines.KNeighborsClassifier static method*), 1712
load() (*evalml.pipelines.LightGBMClassifier static method*), 1716
load() (*evalml.pipelines.LightGBMRegressor static method*), 1719
load() (*evalml.pipelines.LinearRegressor static method*), 1722
load() (*evalml.pipelines.LogisticRegressionClassifier static method*), 1725
load() (*evalml.pipelines.multiclass_classification_pipeline.MulticlassClassificationPipeline static method*), 1579
load() (*evalml.pipelines.MulticlassClassificationPipeline static method*), 1731
load() (*evalml.pipelines.OneHotEncoder static method*), 1735
load() (*evalml.pipelines.OrdinalEncoder static method*), 1739
load() (*evalml.pipelines.PerColumnImputer static method*), 1741
load() (*evalml.pipelines.pipeline_base.PipelineBase static method*), 1585
load() (*evalml.pipelines.PipelineBase static method*), 1745
load() (*evalml.pipelines.ProphetRegressor static method*), 1750
load() (*evalml.pipelines.RandomForestClassifier static method*), 1753
load() (*evalml.pipelines.RandomForestRegressor static method*), 1756
load() (*evalml.pipelines.regression_pipeline.RegressionPipeline static method*), 1593
load() (*evalml.pipelines.RegressionPipeline static method*), 1761
load() (*evalml.pipelines.RFClassifierSelectFromModel static method*), 1765
load() (*evalml.pipelines.RFRegressorSelectFromModel static method*), 1768
load() (*evalml.pipelines.SimpleImputer static method*), 1771
load() (*evalml.pipelines.StackedEnsembleBase static method*), 1774
load() (*evalml.pipelines.StackedEnsembleClassifier static method*), 1778
load() (*evalml.pipelines.StackedEnsembleRegressor static method*), 1782
load() (*evalml.pipelines.StandardScaler static method*), 1784
load() (*evalml.pipelines.SVMClassifier static method*), 1787
load() (*evalml.pipelines.SVMRegressor static method*), 1790
load() (*evalml.pipelines.TargetEncoder static method*), 1793
load() (*evalml.pipelines.time_series_classification_pipelines.TimeSeriesB static method*), 1601
load() (*evalml.pipelines.time_series_classification_pipelines.TimeSeriesC static method*), 1608
load() (*evalml.pipelines.time_series_classification_pipelines.TimeSeriesM static method*), 1616
load() (*evalml.pipelines.time_series_pipeline_base.TimeSeriesPipelineBa static method*), 1624
load() (*evalml.pipelines.time_series_regression_pipeline.TimeSeriesRegre static method*), 1633
load() (*evalml.pipelines.TimeSeriesBinaryClassificationPipeline static method*), 1799
load() (*evalml.pipelines.TimeSeriesClassificationPipeline static method*), 1806
load() (*evalml.pipelines.TimeSeriesFeaturizer static method*), 1811
load() (*evalml.pipelines.TimeSeriesImputer static method*), 1814
load() (*evalml.pipelines.TimeSeriesMulticlassClassificationPipeline static method*), 1820
load() (*evalml.pipelines.TimeSeriesRegressionPipeline static method*), 1829
load() (*evalml.pipelines.TimeSeriesRegularizer static method*), 1833
load() (*evalml.pipelines.Transformer static method*), 1836
load() (*evalml.pipelines.VowpalWabbitBinaryClassifier static method*), 1839
load() (*evalml.pipelines.VowpalWabbitMulticlassClassifier static method*), 1842
load() (*evalml.pipelines.VowpalWabbitRegressor static method*), 1845
load() (*evalml.pipelines.XGBoostClassifier static method*), 1848
load() (*evalml.pipelines.XGBoostRegressor static method*), 1851
load_breast_cancer() (*in module evalml.demos*), 391
load_breast_cancer() (*in module evalml.demos.breast_cancer*), 388
load_churn() (*in module evalml.demos*), 391

load_churn() (*in module evalml.demos.churn*), 388
load_data() (*in module evalml.preprocessing*), 1869
load_data() (*in module evalml.preprocessing.utils*), 1866
load_diabetes() (*in module evalml.demos*), 391
load_diabetes() (*in module evalml.demos.diabetes*), 389
load_fraud() (*in module evalml.demos*), 391
load_fraud() (*in module evalml.demos.fraud*), 389
load_weather() (*in module evalml.demos*), 391
load_weather() (*in module evalml.demos.weather*), 390
load_wine() (*in module evalml.demos*), 391
load_wine() (*in module evalml.demos.wine*), 390
log_batch_times() (*in module evalml.utils.logger*), 1906
log_error_callback() (*in module evalml.automl.callbacks*), 266
log_subtitle() (*in module evalml.utils*), 1913
log_subtitle() (*in module evalml.utils.logger*), 1906
log_title() (*in module evalml.utils*), 1913
log_title() (*in module evalml.utils.logger*), 1906
logger (*in module evalml.automl.callbacks*), 266
logger (*in module evalml.objectives.sensitivity_low_alert*), 459
logger (*in module evalml.pipelines.component_graph*), 1573
logger (*in module evalml.pipelines.pipeline_base*), 1581
logger (*in module evalml.tuners.skopt_tuner*), 1888
logger (*in module evalml.utils.gen_utils*), 1905
LogisticRegressionClassifier (*class in evalml.pipelines*), 1723
LogisticRegressionClassifier (*class in evalml.pipelines.components*), 1425
LogisticRegressionClassifier (*class in evalml.pipelines.components.estimators*), 918
LogisticRegressionClassifier (*class in evalml.pipelines.components.estimators.classifiers*), 720
LogisticRegressionClassifier (*class in evalml.pipelines.components.estimators.classifiers.logistic_regression_classifier*), 671
LogLossBinary (*class in evalml.objectives*), 572
LogLossBinary (*class in evalml.objectives.standard_metrics*), 489
LogLossMulticlass (*class in evalml.objectives*), 574
LogLossMulticlass (*class in evalml.objectives.standard_metrics*), 492
LogTransformer (*class in evalml.pipelines.components*), 1429
LogTransformer (*class in evalml.pipelines.components.transformers*), 1243
LogTransformer (*class in evalml.pipelines.components.transformers.preprocessing*), 1148
LogTransformer (*class in evalml.pipelines.components.transformers.preprocessing.log_tran*, 1088
LSA (*class in evalml.pipelines.components*), 1432
LSA (*class in evalml.pipelines.components.transformers*), 1246
LSA (*class in evalml.pipelines.components.transformers.preprocessing*), 1150
LSA (*class in evalml.pipelines.components.transformers.preprocessing.lsa*), 1091


## M


MAE (*class in evalml.objectives*), 576
MAE (*class in evalml.objectives.standard_metrics*), 493
make_balancing_dictionary() (*in module evalml.pipelines.components.utils*), 1330
make_data_splitter() (*in module evalml.automl*), 280
make_data_splitter() (*in module evalml.automl.utils*), 271
make_pipeline() (*in module evalml.pipelines.utils*), 1637
make_pipeline_from_actions() (*in module evalml.pipelines.utils*), 1638
make_pipeline_from_data_check_output() (*in module evalml.pipelines.utils*), 1638
make_timeseries_baseline_pipeline() (*in module evalml.pipelines.utils*), 1639
MAPE (*class in evalml.objectives*), 578
MAPE (*class in evalml.objectives.standard_metrics*), 495
MaxError (*class in evalml.objectives*), 579
MaxError (*class in evalml.objectives.standard_metrics*), 497
MCCBinary (*class in evalml.objectives*), 581
MCCBinary (*class in evalml.objectives.standard_metrics*), 499
MCCMulticlass (*class in evalml.objectives*), 583
MCCMulticlass (*class in evalml.objectives.standard_metrics*), 501
MeanSquaredLogError (*class in evalml.objectives*), 585
MeanSquaredLogError (*class in evalml.objectives.standard_metrics*), 503
MedianAE (*class in evalml.objectives*), 587
MedianAE (*class in evalml.objectives.standard_metrics*), 504
method (*in module evalml.utils.update_checker*), 1907
MethodPropertyNotFoundError, 392, 395
MissingComponentError, 392, 395
model_family (*evalml.pipelines.binary_classification_pipeline.BinaryCla*, *property*), 1558

model_family (*evalml.pipelines.BinaryClassificationPipeline property*), 1650
model_family (*evalml.pipelines.classification_pipeline.ClassificationPipeline property*), 1566
model_family (*evalml.pipelines.ClassificationPipeline property*), 1663
model_family (*evalml.pipelines.components.Estimator property*), 1390
model_family (*evalml.pipelines.components.estimators.Estimator property*), 892
model_family (*evalml.pipelines.components.estimators.estimator.Estimator property*), 856
model_family (*evalml.pipelines.Estimator property*), 1692
model_family (*evalml.pipelines.multiclass_classification_pipeline.MulticlassClassificationPipeline property*), 1579
model_family (*evalml.pipelines.MulticlassClassificationPipeline property*), 1731
model_family (*evalml.pipelines.pipeline_base.PipelineBase property*), 1586
model_family (*evalml.pipelines.PipelineBase property*), 1746
model_family (*evalml.pipelines.regression_pipeline.RegressionPipeline property*), 1594
model_family (*evalml.pipelines.RegressionPipeline property*), 1762
model_family (*evalml.pipelines.time_series_classification_pipeline.TimeSeriesClassificationPipeline property*), 1601
model_family (*evalml.pipelines.time_series_classification_pipelines.TimeSeriesClassificationPipeline property*), 1608
model_family (*evalml.pipelines.time_series_classification_pipelines.TimeSeriesMulticlassClassificationPipeline property*), 1616
model_family (*evalml.pipelines.time_series_pipeline_base.TimeSeriesPipelineBase property*), 1624
model_family (*evalml.pipelines.time_series_regression_pipeline.TimeSeriesRegressionPipeline property*), 1633
model_family (*evalml.pipelines.TimeSeriesBinaryClassificationPipeline property*), 1799
model_family (*evalml.pipelines.TimeSeriesClassificationPipeline property*), 1806
model_family (*evalml.pipelines.TimeSeriesMulticlassClassificationPipeline property*), 1820
model_family (*evalml.pipelines.TimeSeriesRegressionPipeline property*), 1829
ModelFamily (*class in evalml.model_family*), 400
ModelFamily (*class in evalml.model_family.model_family*), 398
modifies_features (*evalml.pipelines.components.component_base.ComponentBase property*), 1325
modifies_features (*evalml.pipelines.components.ComponentBase property*), 1355
modifies_target (*evalml.pipelines.components.component_base.ComponentBase property*), 1325
modifies_target (*evalml.pipelines.components.ComponentBase property*), 1355
module
evalml, 223
evalml.automl, 223
evalml.automl.automl_algorithm, 223
evalml.automl.automl_algorithm.automl_algorithm, 223
evalml.automl.automl_algorithm.default_algorithm, 225
evalml.automl.automl_algorithm.iterative_algorithm, 228
evalml.automl.automl_search, 257
evalml.automl.callbacks, 266
evalml.automl.engine, 238
evalml.automl.engine.engine, 238
evalml.automl.engine.dask_engine, 241
evalml.automl.engine.engine_base, 244
evalml.automl.engine.sequential_engine, 248
evalml.automl.pipeline_search_plots, 267
evalml.automl.progress, 268
evalml.automl.utils, 269
evalml.data_checks, 285
evalml.data_checks.class_imbalance_data_check, 286
evalml.data_checks.data_check, 289
evalml.data_checks.data_check_action, 289
evalml.data_checks.data_check_action_code, 290
evalml.data_checks.data_check_action_option, 291
evalml.data_checks.data_check_message, 292
evalml.data_checks.data_check_message_code, 295
evalml.data_checks.data_check_message_type, 298
evalml.data_checks.data_checks, 299
evalml.data_checks.datetime_format_data_check, 299
evalml.data_checks.default_data_checks, 307
evalml.data_checks.id_columns_data_check, 308
evalml.data_checks.invalid_target_data_check, 312
evalml.data_checks.multicollinearity_data_check, 316
evalml.data_checks.no_variance_data_check, 318
evalml.data_checks.null_data_check, 321
evalml.data_checks.outliers_data_check, 325
evalml.data_checks.sparsity_data_check,

328
evalml.data_checks.target_distribution_data_check, 330
evalml.data_checks.target_leakage_data_check, 332
evalml.data_checks.ts_parameters_data_check, 334
evalml.data_checks.ts_splitting_data_check, 336
evalml.data_checks.uniqueness_data_check, 338
evalml.data_checks.utils, 340
evalml.demos, 387
evalml.demos.breast_cancer, 387
evalml.demos.churn, 388
evalml.demos.diabetes, 388
evalml.demos.fraud, 389
evalml.demos.weather, 389
evalml.demos.wine, 390
evalml.exceptions, 392
evalml.exceptions.exceptions, 392
evalml.model_family, 398
evalml.model_family.model_family, 398
evalml.model_family.utils, 399
evalml.model_understanding, 402
evalml.model_understanding.decision_boundary, 409
evalml.model_understanding.feature_explanations, 410
evalml.model_understanding.force_plots, 411
evalml.model_understanding.metrics, 413
evalml.model_understanding.partial_dependence_plots, 416
evalml.model_understanding.permutation_importance, 418
evalml.model_understanding.prediction_explanations, 402
evalml.model_understanding.prediction_explanations.explainers, 402
evalml.model_understanding.visualizations, 420
evalml.objectives, 440
evalml.objectives.binary_classification_objective, 440
evalml.objectives.cost_benefit_matrix, 443
evalml.objectives.fraud_cost, 446
evalml.objectives.lead_scoring, 449
evalml.objectives.multiclass_classification_objective, 452
evalml.objectives.objective_base, 454
evalml.objectives.regression_objective, 457
evalml.objectives.sensitivity_low_alert, 459
evalml.objectives.standard_metrics, 462
evalml.objectives.time_series_regression_objective, 527
evalml.objectives.utils, 530
evalml.pipelines, 619
evalml.pipelines.binary_classification_pipeline, 1552
evalml.pipelines.binary_classification_pipeline_mixin, 1560
evalml.pipelines.classification_pipeline, 1561
evalml.pipelines.component_graph, 1568
evalml.pipelines.components, 619
evalml.pipelines.components.component_base, 1323
evalml.pipelines.components.component_base_meta, 1326
evalml.pipelines.components.ensemble, 619
evalml.pipelines.components.ensemble.stacked_ensemble_ 619
evalml.pipelines.components.ensemble.stacked_ensemble_ 622
evalml.pipelines.components.ensemble.stacked_ensemble_ 627
evalml.pipelines.components.estimators, 642
evalml.pipelines.components.estimators.classifiers, 642
evalml.pipelines.components.estimators.classifiers.bas 642
evalml.pipelines.components.estimators.classifiers.cat 646
evalml.pipelines.components.estimators.classifiers.dec 650
evalml.pipelines.components.estimators.classifiers.ela 654
evalml.pipelines.components.estimators.classifiers.et_ 658
evalml.pipelines.components.estimators.classifiers.kne 663
evalml.pipelines.components.estimators.classifiers.lig 667
evalml.pipelines.components.estimators.classifiers.log 671
evalml.pipelines.components.estimators.classifiers.rf_ 675
evalml.pipelines.components.estimators.classifiers.svm 678
evalml.pipelines.components.estimators.classifiers.vow 682
evalml.pipelines.components.estimators.classifiers.xgb 692

evalml.pipelines.components.estimators.estimator, 853
evalml.pipelines.components.estimators.regressor, 741
evalml.pipelines.components.estimators.regressor.arima_regressor, 741
evalml.pipelines.components.estimators.regressor.baseline_regressor, 746
evalml.pipelines.components.estimators.regressor.catboost_regressor, 749
evalml.pipelines.components.estimators.regressor.decision_tree_regressor, 753
evalml.pipelines.components.estimators.regressor.elastic_net_regressor, 758
evalml.pipelines.components.estimators.regressor.et_regressor, 761
evalml.pipelines.components.estimators.regressor.exponential_smoothing_regressor, 766
evalml.pipelines.components.estimators.regressor.lightgbm_regressor, 770
evalml.pipelines.components.estimators.regressor.linear_regressor, 774
evalml.pipelines.components.estimators.regressor.prophet_regressor, 777
evalml.pipelines.components.estimators.regressor.rf_regressor, 782
evalml.pipelines.components.estimators.regressor.svm_regressor, 786
evalml.pipelines.components.estimators.regressor.time_series_baseline_estimator, 790
evalml.pipelines.components.estimators.regressor.xgb_regressor, 793
evalml.pipelines.components.estimators.regressor.xgboost_regressor, 797
evalml.pipelines.components.transformers, 956
evalml.pipelines.components.transformers.column_selectors, 1201
evalml.pipelines.components.transformers.dimensionality_reduction, 957
evalml.pipelines.components.transformers.dimensionality_reduction.lda, 957
evalml.pipelines.components.transformers.dimensionality_reduction.pca, 960
evalml.pipelines.components.transformers.encoder, 968
evalml.pipelines.components.transformers.encoder.base_encoder, 968
evalml.pipelines.components.transformers.encoder.one_hot_encoder, 971
evalml.pipelines.components.transformers.encoder.ordinal_encoder, 976
evalml.pipelines.components.transformers.encoder.target_encoder, 980
evalml.pipelines.components.transformers.feature_selection, 996
evalml.pipelines.components.transformers.feature_selection.base_feature_selector, 996
evalml.pipelines.components.transformers.feature_selection.rfecv_feature_selector, 999
evalml.pipelines.components.transformers.feature_selection.select_columns_transformer, 1009
evalml.pipelines.components.transformers.feature_selection.variance_threshold_selector, 1013
evalml.pipelines.components.transformers.imputers, 1032
evalml.pipelines.components.transformers.imputers.imputer, 1032
evalml.pipelines.components.transformers.imputers.knn_imputer, 1036
evalml.pipelines.components.transformers.imputers.per_column_imputer, 1039
evalml.pipelines.components.transformers.imputers.simple_imputer, 1042
evalml.pipelines.components.transformers.imputers.target_imputer, 1045
evalml.pipelines.components.transformers.imputers.time_series_imputer, 1048
evalml.pipelines.components.transformers.preprocessing, 1068
evalml.pipelines.components.transformers.preprocessing.delay_target_transformer, 1068
evalml.pipelines.components.transformers.preprocessing.drop_columns_transformer, 1071
evalml.pipelines.components.transformers.preprocessing.drop_nan_rows_transformer, 1076
evalml.pipelines.components.transformers.preprocessing.lagged_feature_extractor, 1079
evalml.pipelines.components.transformers.preprocessing.polynomial_decomposer, 1082
evalml.pipelines.components.transformers.preprocessing.polynomial_detrender, 1085
evalml.pipelines.components.transformers.preprocessing.text_transformer, 1088
evalml.pipelines.components.transformers.preprocessing.transformer, 1091
evalml.pipelines.components.transformers.preprocessing.transformer_base, 1094
evalml.pipelines.components.transformers.preprocessing.utils, 1097
evalml.pipelines.components.transformers.preprocessing.label_encoder, 1103
evalml.pipelines.components.transformers.preprocessing.one_hot_encoder, 1106
evalml.pipelines.components.transformers.preprocessing.ordinal_encoder, 1112
evalml.pipelines.components.transformers.preprocessing.target_encoder, 1115

evalml.pipelines.components.transformers.preprocessing.imbalanced_data_transformer, 1119
evalml.pipelines.components.transformers.preprocessing.text_transformer, 1122
evalml.pipelines.components.transformers.samplers, 1180
evalml.pipelines.components.transformers.samplers.base_sampler, 1180
evalml.pipelines.components.transformers.samplers.oversampler, 1183
evalml.pipelines.components.transformers.samplers.undersampler, 1186
evalml.pipelines.components.transformers.scalers, 1195
evalml.pipelines.components.transformers.scalers.standard_scaler, 1195
evalml.pipelines.components.transformers.transformer, 1211
evalml.pipelines.components.utils, 1327
evalml.pipelines.multiclass_classification_pipeline, 1574
evalml.pipelines.pipeline_base, 1581
evalml.pipelines.pipeline_meta, 1587
evalml.pipelines.regression_pipeline, 1589
evalml.pipelines.time_series_classification_pipelines, 1595
evalml.pipelines.time_series_pipeline_base, 1619
evalml.pipelines.time_series_regression_pipeline, 1626
evalml.pipelines.utils, 1635
evalml.preprocessing, 1852
evalml.preprocessing.data_splitters, 1852
evalml.preprocessing.data_splitters.no_split, 1853
evalml.preprocessing.data_splitters.sk_splitters, 1854
evalml.preprocessing.data_splitters.time_series_split, 1856
evalml.preprocessing.data_splitters.training_validation_split, 1858
evalml.preprocessing.utils, 1866
evalml.problem_types, 1875
evalml.problem_types.problem_types, 1876
evalml.problem_types.utils, 1877
evalml.tuners, 1883
evalml.tuners.grid_search_tuner, 1883
evalml.tuners.random_search_tuner, 1885
evalml.tuners.skopt_tuner, 1887
evalml.tuners.tuner, 1889
evalml.tuners.tuner_exceptions, 1891
evalml.utils, 1897
evalml.utils.base_meta, 1897
evalml.utils.cli_utils, 1898
evalml.utils.gen_utils, 1900
evalml.utils.logger, 1906
evalml.utils.nullable_type_utils, 1907
evalml.utils.update_checker, 1907
evalml.utils.woodwork_utils, 1907
MSE (*class in evalml.objectives*), 588
MSE (*class in evalml.objectives.standard_metrics*), 506
MulticlassClassificationObjective (*class in evalml.objectives*), 590
MulticlassClassificationObjective (*class in evalml.objectives.multiclass_classification_objective*), 452
MulticlassClassificationPipeline (*class in evalml.pipelines*), 1726
MulticlassClassificationPipeline (*class in evalml.pipelines.multiclass_classification_pipeline*), 1574
MulticollinearityDataCheck (*class in evalml.data_checks*), 368
MulticollinearityDataCheck (*class in evalml.data_checks.multicollinearity_data_check*), 317

## N

name (*evalml.objectives.binary_classification_objective.BinaryClassificatio property*), 442
name (*evalml.objectives.BinaryClassificationObjective property*), 551
name (*evalml.objectives.multiclass_classification_objective.MulticlassClass property*), 453
name (*evalml.objectives.MulticlassClassificationObjective property*), 591
name (*evalml.objectives.objective_base.ObjectiveBase property*), 455
name (*evalml.objectives.ObjectiveBase property*), 593
name (*evalml.objectives.regression_objective.RegressionObjective property*), 458
name (*evalml.objectives.RegressionObjective property*), 612
name (*evalml.objectives.time_series_regression_objective.TimeSeriesRegres property*), 529
name (*evalml.pipelines.binary_classification_pipeline.BinaryClassificationI property*), 1558
name (*evalml.pipelines.BinaryClassificationPipeline property*), 1650
name (*evalml.pipelines.classification_pipeline.ClassificationPipeline property*), 1566
name (*evalml.pipelines.ClassificationPipeline property*), 1663
name (*evalml.pipelines.components.component_base.ComponentBase property*), 1325
name (*evalml.pipelines.components.ComponentBase property*), 1355

name (evalml.pipelines.components.ensemble.stacked_ensemble_base.StackedEnsembleBase property), 621
name (evalml.pipelines.components.ensemble.StackedEnsembleBase property), 633
name (evalml.pipelines.components.Estimator property), 1390
name (evalml.pipelines.components.estimators.classifiers.vowpal_wabbit_classifiers.VowpalWabbitBaseClassifier property), 684
name (evalml.pipelines.components.estimators.Estimator property), 892
name (evalml.pipelines.components.estimators.estimator.Estimator property), 856
name (evalml.pipelines.components.FeatureSelector property), 1403
name (evalml.pipelines.components.StackedEnsembleBase property), 1489
name (evalml.pipelines.components.Transformer property), 1531
name (evalml.pipelines.components.transformers.column_selectors.ColumnSelector property), 1203
name (evalml.pipelines.components.transformers.feature_selection.feature_selection_base.FeatureSelectionBase property), 998
name (evalml.pipelines.components.transformers.feature_selection.FeatureSelection property), 1018
name (evalml.pipelines.components.transformers.feature_selection.recursive_feature_elimination_cv_selector.RecursiveFeatureEliminationCV property), 1002
name (evalml.pipelines.components.transformers.FeatureSelector property), 1235
name (evalml.pipelines.components.transformers.preprocessing.text_transformer.TextTransformer property), 1114
name (evalml.pipelines.components.transformers.preprocessing.transformer.PreprocessingTransformer property), 1170
name (evalml.pipelines.components.transformers.samplers.base_sampler.BaseSampler property), 1182
name (evalml.pipelines.components.transformers.Transformer property), 1317
name (evalml.pipelines.components.transformers.transformer.Transformer property), 1213
name (evalml.pipelines.Estimator property), 1692
name (evalml.pipelines.FeatureSelector property), 1706
name (evalml.pipelines.multiclass_classification_pipeline.MulticlassClassificationPipeline property), 1579
name (evalml.pipelines.MulticlassClassificationPipeline property), 1731
name (evalml.pipelines.pipeline_base.PipelineBase property), 1586
name (evalml.pipelines.PipelineBase property), 1746
name (evalml.pipelines.regression_pipeline.RegressionPipeline property), 1594
name (evalml.pipelines.RegressionPipeline property), 1762
name (evalml.pipelines.StackedEnsembleBase property), 1774
name (evalml.pipelines.time_series_classification_pipelines.TimeSeriesBinaryClassificationPipeline property), 1601
name (evalml.pipelines.time_series_classification_pipelines.TimeSeriesClassificationPipeline property), 1609
name (evalml.pipelines.time_series_classification_pipelines.TimeSeriesMulticlassClassificationPipeline property), 1616
name (evalml.pipelines.time_series_pipeline_base.TimeSeriesPipelineBase property), 1624
name (evalml.pipelines.time_series_regression_pipeline.TimeSeriesRegressionPipeline property), 1633
name (evalml.pipelines.TimeSeriesBinaryClassificationPipeline property), 1799
name (evalml.pipelines.TimeSeriesClassificationPipeline property), 1806
name (evalml.pipelines.TimeSeriesMulticlassClassificationPipeline property), 1820
name (evalml.pipelines.TimeSeriesRegressionPipeline property), 1829
name (evalml.pipelines.Transformer property), 1836
name() (evalml.data_checks.class_imbalance_data_check.ClassImbalanceDataCheck method), 287
name() (evalml.data_checks.ClassImbalanceDataCheck method), 343
name() (evalml.data_checks.data_check.DataCheck method), 289
name() (evalml.data_checks.data_check_action_code.DataCheckActionCode method), 291
name() (evalml.data_checks.data_check_action_option.DCAOParameterAllowedValuesType method), 293
name() (evalml.data_checks.data_check_action_option.DCAOParameterType method), 294
name() (evalml.data_checks.data_check_message_code.DataCheckMessageCode method), 298
name() (evalml.data_checks.data_check_message_type.DataCheckMessageType method), 298
name() (evalml.data_checks.DataCheck method), 345
name() (evalml.data_checks.DataCheckActionCode method), 346
name() (evalml.data_checks.DataCheckMessageCode method), 350
name() (evalml.data_checks.DataCheckMessageType method), 350
name() (evalml.data_checks.datetime_format_data_check.DateTimeFormatDataCheck method), 300
name() (evalml.data_checks.DateTimeFormatDataCheck method), 352
name() (evalml.data_checks.DCAOParameterAllowedValuesType method), 359
name() (evalml.data_checks.DCAOParameterType method), 359
name() (evalml.data_checks.id_columns_data_check.IDColumnsDataCheck method), 309
name() (evalml.data_checks.IDColumnsDataCheck method), 361
name() (evalml.data_checks.invalid_target_data_check.InvalidTargetDataC

*method*), 313
name() (*evalml.data_checks.InvalidTargetDataCheck method*), 365
name() (*evalml.data_checks.multicollinearity_data_check.MulticollinearityDataCheck method*), 317
name() (*evalml.data_checks.MulticollinearityDataCheck method*), 368
name() (*evalml.data_checks.no_variance_data_check.NoVarianceDataCheck method*), 318
name() (*evalml.data_checks.NoVarianceDataCheck method*), 369
name() (*evalml.data_checks.null_data_check.NullDataCheck method*), 322
name() (*evalml.data_checks.NullDataCheck method*), 372
name() (*evalml.data_checks.outliers_data_check.OutliersDataCheck method*), 326
name() (*evalml.data_checks.OutliersDataCheck method*), 376
name() (*evalml.data_checks.sparsity_data_check.SparsityDataCheck method*), 328
name() (*evalml.data_checks.SparsityDataCheck method*), 378
name() (*evalml.data_checks.target_distribution_data_check.TargetDistributionDataCheck method*), 330
name() (*evalml.data_checks.target_leakage_data_check.TargetLeakageDataCheck method*), 332
name() (*evalml.data_checks.TargetDistributionDataCheck method*), 379
name() (*evalml.data_checks.TargetLeakageDataCheck method*), 381
name() (*evalml.data_checks.TimeSeriesParametersDataCheck method*), 383
name() (*evalml.data_checks.TimeSeriesSplittingDataCheck method*), 384
name() (*evalml.data_checks.ts_parameters_data_check.TimeSeriesParametersDataCheck method*), 335
name() (*evalml.data_checks.ts_splitting_data_check.TimeSeriesSplittingDataCheck method*), 336
name() (*evalml.data_checks.uniqueness_data_check.UniquenessDataCheck method*), 338
name() (*evalml.data_checks.UniquenessDataCheck method*), 385
name() (*evalml.exceptions.exceptions.PartialDependenceErrorCode method*), 393
name() (*evalml.exceptions.exceptions.PipelineErrorCodeEnum method*), 394
name() (*evalml.exceptions.exceptions.ValidationErrorCode method*), 395
name() (*evalml.exceptions.PartialDependenceErrorCode method*), 396
name() (*evalml.exceptions.PipelineErrorCodeEnum method*), 397
name() (*evalml.exceptions.ValidationErrorCode method*), 397
name() (*evalml.model_family.model_family.ModelFamily method*), 399
name() (*evalml.model_family.ModelFamily method*), 401
name() (*evalml.model_understanding.prediction_explanations.explainers.E method*), 406
name() (*evalml.problem_types.problem_types.ProblemTypes method*), 1876
name() (*evalml.problem_types.ProblemTypes method*), 1883
NaturalLanguageFeaturizer (*class in evalml.pipelines.components*), 1434
NaturalLanguageFeaturizer (*class in evalml.pipelines.components.transformers*), 1248
NaturalLanguageFeaturizer (*class in evalml.pipelines.components.transformers.preprocessing*), 1152
NaturalLanguageFeaturizer (*class in evalml.pipelines.components.transformers.preprocessing.natural_ 1094*
needs_fitting() (*evalml.pipelines.ARIMARegressor method*), 1644
needs_fitting() (*evalml.pipelines.CatBoostClassifier method*), 1655
needs_fitting() (*evalml.pipelines.CatBoostRegressor method*), 1658
needs_fitting() (*evalml.pipelines.components.ARIMARegressor method*), 1339
needs_fitting() (*evalml.pipelines.components.BaselineClassifier method*), 1343
needs_fitting() (*evalml.pipelines.components.BaselineRegressor method*), 1346
needs_fitting() (*evalml.pipelines.components.CatBoostClassifier method*), 1349
needs_fitting() (*evalml.pipelines.components.CatBoostRegressor method*), 1352
needs_fitting() (*evalml.pipelines.components.component_base.Compon method*), 1325
needs_fitting() (*evalml.pipelines.components.ComponentBase method*), 1355
needs_fitting() (*evalml.pipelines.components.DateTimeFeaturizer method*), 1358
needs_fitting() (*evalml.pipelines.components.DecisionTreeClassifier method*), 1362
needs_fitting() (*evalml.pipelines.components.DecisionTreeRegressor method*), 1365
needs_fitting() (*evalml.pipelines.components.DFSTransformer method*), 1368
needs_fitting() (*evalml.pipelines.components.DropNaNRowsTransform method*), 1373
needs_fitting() (*evalml.pipelines.components.DropNullColumns method*), 1376
needs_fitting() (*evalml.pipelines.components.DropRowsTransformer*

*method*), 1378
needs_fitting() (*evalml.pipelines.components.ElasticNetClassifier*
*method*), 1381
needs_fitting() (*evalml.pipelines.components.ElasticNetRegressor*
*method*), 1384
needs_fitting() (*evalml.pipelines.components.EmailFeaturesTransformer*
*method*), 1387
needs_fitting() (*evalml.pipelines.components.ensemble.StackedEnsembleBase*
*method*), 621
needs_fitting() (*evalml.pipelines.components.ensemble.StackedEnsembleClassifier*
*method*), 626
needs_fitting() (*evalml.pipelines.components.ensemble.StackedEnsembleRegressor*
*method*), 630
needs_fitting() (*evalml.pipelines.components.ensemble.StacksEnsemble*
*method*), 633
needs_fitting() (*evalml.pipelines.components.ensemble.VotingEnsembleClassifier*
*method*), 637
needs_fitting() (*evalml.pipelines.components.ensemble.VotingEnsembleRegressor*
*method*), 641
needs_fitting() (*evalml.pipelines.components.Estimator*
*method*), 1390
needs_fitting() (*evalml.pipelines.components.estimators.ARIMARegressor*
*method*), 861
needs_fitting() (*evalml.pipelines.components.estimators.BaselineClassifier*
*method*), 865
needs_fitting() (*evalml.pipelines.components.estimators.BaselineRegressor*
*method*), 868
needs_fitting() (*evalml.pipelines.components.estimators.CatBoostClassifier*
*method*), 871
needs_fitting() (*evalml.pipelines.components.estimators.CatBoostRegressor*
*method*), 874
needs_fitting() (*evalml.pipelines.components.estimators.classifiers.BaselineClassifier*
*method*), 645
needs_fitting() (*evalml.pipelines.components.estimators.classifiers.CatBoostClassifier*
*method*), 698
needs_fitting() (*evalml.pipelines.components.estimators.classifiers.ElasticNetClassifier*
*method*), 649
needs_fitting() (*evalml.pipelines.components.estimators.classifiers.EvalMLPipelineComponentClassifier*
*method*), 702
needs_fitting() (*evalml.pipelines.components.estimators.classifiers.DecisionTreeClassifier*
*method*), 653
needs_fitting() (*evalml.pipelines.components.estimators.classifiers.ExtraTreesClassifier*
*method*), 705
needs_fitting() (*evalml.pipelines.components.estimators.classifiers.KNeighborsClassifier*
*method*), 657
needs_fitting() (*evalml.pipelines.components.estimators.classifiers.LightGBMClassifier*
*method*), 709
needs_fitting() (*evalml.pipelines.components.estimators.classifiers.LinearDiscriminantAnalysis*
*method*), 661
needs_fitting() (*evalml.pipelines.components.estimators.classifiers.LinearSVCClassifier*
*method*), 712
needs_fitting() (*evalml.pipelines.components.estimators.classifiers.LogisticRegressionClassifier*
*method*), 666
needs_fitting() (*evalml.pipelines.components.estimators.classifiers.RandomForestClassifier*
*method*), 716
needs_fitting() (*evalml.pipelines.components.estimators.classifiers.lightgbm_dart_classifier.LightGBMDartClassifier*
*method*), 670
needs_fitting() (*evalml.pipelines.components.estimators.classifiers.LightGBMClassifier*
*method*), 720
needs_fitting() (*evalml.pipelines.components.estimators.classifiers.logistic_regression_classifier.LogisticRegressionClassifier*
*method*), 673
needs_fitting() (*evalml.pipelines.components.estimators.classifiers.LogisticRegressionCV*
*method*), 723
needs_fitting() (*evalml.pipelines.components.estimators.classifiers.RandomForestClassifier*
*method*), 726
needs_fitting() (*evalml.pipelines.components.estimators.classifiers.rf_classifier.RandomForestClassifier*
*method*), 677
needs_fitting() (*evalml.pipelines.components.estimators.classifiers.svm_classifier.SVMClassifier*
*method*), 681
needs_fitting() (*evalml.pipelines.components.estimators.classifiers.SVMClassifier*
*method*), 730
needs_fitting() (*evalml.pipelines.components.estimators.classifiers.vowpal_base.VowpalBaseClassifier*
*method*), 684
needs_fitting() (*evalml.pipelines.components.estimators.classifiers.vowpal_base.VowpalBaseRegressor*
*method*), 688
needs_fitting() (*evalml.pipelines.components.estimators.classifiers.vowpal_base.VowpalBaseRegressor.VowpalBaseRegressor*
*method*), 691
needs_fitting() (*evalml.pipelines.components.estimators.classifiers.VowpalBaseClassifier*
*method*), 733
needs_fitting() (*evalml.pipelines.components.estimators.classifiers.VowpalBaseRegressor*
*method*), 736
needs_fitting() (*evalml.pipelines.components.estimators.classifiers.xgboost_classifier.XGBoostClassifier*
*method*), 695
needs_fitting() (*evalml.pipelines.components.estimators.classifiers.XGBoostRegressor*
*method*), 740
needs_fitting() (*evalml.pipelines.components.estimators.DecisionTreeClassifier*
*method*), 878
needs_fitting() (*evalml.pipelines.components.estimators.DecisionTreeRegressor*
*method*), 882
needs_fitting() (*evalml.pipelines.components.estimators.ElasticNetClassifier*
*method*), 886
needs_fitting() (*evalml.pipelines.components.estimators.ElasticNetRegressor*
*method*), 889
needs_fitting() (*evalml.pipelines.components.estimators.Estimator*
*method*), 892
needs_fitting() (*evalml.pipelines.components.estimator.Estimator*
*method*), 856
needs_fitting() (*evalml.pipelines.components.estimators.ExponentialSmoothing*
*method*), 895
needs_fitting() (*evalml.pipelines.components.estimators.ExtraTreesClassifier*
*method*), 899
needs_fitting() (*evalml.pipelines.components.estimators.ExtraTreesRegressor*
*method*), 903
needs_fitting() (*evalml.pipelines.components.estimators.KNeighborsClassifier*
*method*), 907
needs_fitting() (*evalml.pipelines.components.estimators.LightGBMClassifier*
*method*), 911
needs_fitting() (*evalml.pipelines.components.estimators.LightGBMRegressor*

*method*), 914
needs_fitting() (*evalml.pipelines.components.estimators.*
*method*), 917
needs_fitting() (*evalml.pipelines.components.estimators.*
*method*), 921
needs_fitting() (*evalml.pipelines.components.estimators.*
*method*), 924
needs_fitting() (*evalml.pipelines.components.estimators.*
*method*), 927
needs_fitting() (*evalml.pipelines.components.estimators.*
*method*), 930
needs_fitting() (*evalml.pipelines.components.estimators.*
*method*), 745
needs_fitting() (*evalml.pipelines.components.estimators.*
*method*), 804
needs_fitting() (*evalml.pipelines.components.estimators.*
*method*), 749
needs_fitting() (*evalml.pipelines.components.estimators.*
*method*), 808
needs_fitting() (*evalml.pipelines.components.estimators.*
*method*), 752
needs_fitting() (*evalml.pipelines.components.estimators.*
*method*), 811
needs_fitting() (*evalml.pipelines.components.estimators.*
*method*), 756
needs_fitting() (*evalml.pipelines.components.estimators.*
*method*), 815
needs_fitting() (*evalml.pipelines.components.estimators.*
*method*), 760
needs_fitting() (*evalml.pipelines.components.estimators.*
*method*), 818
needs_fitting() (*evalml.pipelines.components.estimators.*
*method*), 765
needs_fitting() (*evalml.pipelines.components.estimators.*
*method*), 769
needs_fitting() (*evalml.pipelines.components.estimators.*
*method*), 822
needs_fitting() (*evalml.pipelines.components.estimators.*
*method*), 825
needs_fitting() (*evalml.pipelines.components.estimators.*
*method*), 773
needs_fitting() (*evalml.pipelines.components.estimators.*
*method*), 829
needs_fitting() (*evalml.pipelines.components.estimators.*
*method*), 776
needs_fitting() (*evalml.pipelines.components.estimators.*
*method*), 832
needs_fitting() (*evalml.pipelines.components.estimators.*
*method*), 781
needs_fitting() (*evalml.pipelines.components.estimators.*
*method*), 836
needs_fitting() (*evalml.pipelines.components.estimators.*
*method*), 839
needs_fitting() (*evalml.pipelines.components.estimators.*
*method*), 785
needs_fitting() (*evalml.pipelines.components.estimators.regressors.svm*
*method*), 788
needs_fitting() (*evalml.pipelines.components.estimators.regressors.SVI*
*method*), 842
needs_fitting() (*evalml.pipelines.components.estimators.regressors.tim*
*method*), 792
needs_fitting() (*evalml.pipelines.components.estimators.regressors.Tim*
*method*), 846
needs_fitting() (*evalml.pipelines.components.estimators.regressors.vow*
*method*), 796
needs_fitting() (*evalml.pipelines.components.estimators.regressors.Vow*
*method*), 849
needs_fitting() (*evalml.pipelines.components.estimators.regressors.xgb*
*method*), 799
needs_fitting() (*evalml.pipelines.components.estimators.regressors.XG*
*method*), 852
needs_fitting() (*evalml.pipelines.components.estimators.SVMClassifier*
*method*), 933
needs_fitting() (*evalml.pipelines.components.estimators.SVMRegressor*
*method*), 937
needs_fitting() (*evalml.pipelines.components.estimators.TimeSeriesBas*
*method*), 940
needs_fitting() (*evalml.pipelines.components.estimators.VowpalWabbit*
*method*), 943
needs_fitting() (*evalml.pipelines.components.estimators.VowpalWabbit*
*method*), 946
needs_fitting() (*evalml.pipelines.components.estimators.VowpalWabbit*
*method*), 949
needs_fitting() (*evalml.pipelines.components.estimators.XGBoostClass*
*method*), 953
needs_fitting() (*evalml.pipelines.components.estimators.XGBoostRegre*
*method*), 955
needs_fitting() (*evalml.pipelines.components.ExponentialSmoothingRe*
*method*), 1393
needs_fitting() (*evalml.pipelines.components.ExtraTreesClassifier*
*method*), 1397
needs_fitting() (*evalml.pipelines.components.ExtraTreesRegressor*
*method*), 1401
needs_fitting() (*evalml.pipelines.components.LinearFeatureSelector*
*method*), 1403
needs_fitting() (*evalml.pipelines.components.Imputer*
*method*), 1406
needs_fitting() (*evalml.pipelines.components.KNeighborsClassifier*
*method*), 1409
needs_fitting() (*evalml.pipelines.components.LabelEncoder*
*method*), 1412
needs_fitting() (*evalml.pipelines.components.LightGBMClassifier*
*method*), 1416
needs_fitting() (*evalml.pipelines.components.LightGBMRegressor*
*method*), 1419
needs_fitting() (*evalml.pipelines.components.LinearDiscriminantAnaly*
*method*), 1422
needs_fitting() (*evalml.pipelines.components.LinearRegressor*

*method*), 1425
needs_fitting() (*evalml.pipelines.components.LogisticRegressionClassifier* *method*), 1428
needs_fitting() (*evalml.pipelines.components.LogTransformer* *method*), 1431
needs_fitting() (*evalml.pipelines.components.LSA* *method*), 1433
needs_fitting() (*evalml.pipelines.components.NaturalLanguageFeaturizer* *method*), 1436
needs_fitting() (*evalml.pipelines.components.OneHotEncoder* *method*), 1439
needs_fitting() (*evalml.pipelines.components.OrdinalEncoder* *method*), 1442
needs_fitting() (*evalml.pipelines.components.Oversampler* *method*), 1445
needs_fitting() (*evalml.pipelines.components.PCA* *method*), 1447
needs_fitting() (*evalml.pipelines.components.PerColumnImputer* *method*), 1450
needs_fitting() (*evalml.pipelines.components.ProphetRegressor* *method*), 1458
needs_fitting() (*evalml.pipelines.components.RandomForestClassifier* *method*), 1461
needs_fitting() (*evalml.pipelines.components.RandomForestRegressor* *method*), 1464
needs_fitting() (*evalml.pipelines.components.ReplaceNullableTypes* *method*), 1467
needs_fitting() (*evalml.pipelines.components.RFClassifierRFESelector* *method*), 1470
needs_fitting() (*evalml.pipelines.components.RFClassifierSelectFromModel* *method*), 1473
needs_fitting() (*evalml.pipelines.components.RFRegressorRFESelector* *method*), 1476
needs_fitting() (*evalml.pipelines.components.RFRegressorSelectFromModel* *method*), 1479
needs_fitting() (*evalml.pipelines.components.SimpleImputer* *method*), 1486
needs_fitting() (*evalml.pipelines.components.StackedEnsembleBaseClassifier* *method*), 1489
needs_fitting() (*evalml.pipelines.components.StackedEnsembleBaseRegressor* *method*), 1493
needs_fitting() (*evalml.pipelines.components.StackedEnsembleClassifier* *method*), 1497
needs_fitting() (*evalml.pipelines.components.StandardScaler* *method*), 1499
needs_fitting() (*evalml.pipelines.components.SVMClassifier* *method*), 1507
needs_fitting() (*evalml.pipelines.components.SVMRegressor* *method*), 1511
needs_fitting() (*evalml.pipelines.components.TargetEncoder* *method*), 1513
needs_fitting() (*evalml.pipelines.components.TargetImputer* *method*), 1516
needs_fitting() (*evalml.pipelines.components.TimeSeriesBaselineEstimator* *method*), 1519
needs_fitting() (*evalml.pipelines.components.TimeSeriesImputer* *method*), 1525
needs_fitting() (*evalml.pipelines.components.TimeSeriesRegularizer* *method*), 1528
needs_fitting() (*evalml.pipelines.components.Transformer* *method*), 1531
needs_fitting() (*evalml.pipelines.components.transformers.column_sele* *method*), 1203
needs_fitting() (*evalml.pipelines.components.transformers.DateTimeFe* *method*), 1217
needs_fitting() (*evalml.pipelines.components.transformers.DFSTransfo* *method*), 1220
needs_fitting() (*evalml.pipelines.components.transformers.dimensional* *method*), 959
needs_fitting() (*evalml.pipelines.components.transformers.dimensional* *method*), 965
needs_fitting() (*evalml.pipelines.components.transformers.dimensional* *method*), 967
needs_fitting() (*evalml.pipelines.components.transformers.dimensional* *method*), 962
needs_fitting() (*evalml.pipelines.components.transformers.DropNaNRo* *method*), 1225
needs_fitting() (*evalml.pipelines.components.transformers.DropNullCo* *method*), 1227
needs_fitting() (*evalml.pipelines.components.transformers.DropRowsT* *method*), 1230
needs_fitting() (*evalml.pipelines.components.transformers.EmailFeatu* *method*), 1232
needs_fitting() (*evalml.pipelines.components.transformers.encoders.la* *method*), 970
needs_fitting() (*evalml.pipelines.components.transformers.encoders.La* *method*), 986
needs_fitting() (*evalml.pipelines.components.transformers.encoders.on* *method*), 974
needs_fitting() (*evalml.pipelines.components.transformers.encoders.On* *method*), 989
needs_fitting() (*evalml.pipelines.components.transformers.encoders.or* *method*), 979
needs_fitting() (*evalml.pipelines.components.transformers.encoders.Or* *method*), 993
needs_fitting() (*evalml.pipelines.components.transformers.encoders.ta* *method*), 983
needs_fitting() (*evalml.pipelines.components.transformers.encoders.Ta* *method*), 995
needs_fitting() (*evalml.pipelines.components.transformers.feature_sele* *method*), 999
needs_fitting() (*evalml.pipelines.components.transformers.feature_sele* *method*), 1018
needs_fitting() (*evalml.pipelines.components.transformers.feature_sele* *method*), 1002
needs_fitting() (*evalml.pipelines.components.transformers.feature_sele* *method*), 1005
needs_fitting() (*evalml.pipelines.components.transformers.feature_sele*

*method*), 1008
needs_fitting() (*evalml.pipelines.components.transformers.*
*method*), 1012
needs_fitting() (*evalml.pipelines.components.transformers.*
*method*), 1015
needs_fitting() (*evalml.pipelines.components.transformers.*
*method*), 1022
needs_fitting() (*evalml.pipelines.components.transformers.*
*method*), 1025
needs_fitting() (*evalml.pipelines.components.transformers.*
*method*), 1028
needs_fitting() (*evalml.pipelines.components.transformers.*
*method*), 1031
needs_fitting() (*evalml.pipelines.components.transformers.*
*method*), 1235
needs_fitting() (*evalml.pipelines.components.transformers.*
*method*), 1238
needs_fitting() (*evalml.pipelines.components.transformers.*
*method*), 1054
needs_fitting() (*evalml.pipelines.components.transformers.*
*method*), 1035
needs_fitting() (*evalml.pipelines.components.transformers.*
*method*), 1038
needs_fitting() (*evalml.pipelines.components.transformers.*
*method*), 1056
needs_fitting() (*evalml.pipelines.components.transformers.*
*method*), 1041
needs_fitting() (*evalml.pipelines.components.transformers.*
*method*), 1059
needs_fitting() (*evalml.pipelines.components.transformers.*
*method*), 1044
needs_fitting() (*evalml.pipelines.components.transformers.*
*method*), 1061
needs_fitting() (*evalml.pipelines.components.transformers.*
*method*), 1047
needs_fitting() (*evalml.pipelines.components.transformers.*
*method*), 1064
needs_fitting() (*evalml.pipelines.components.transformers.*
*method*), 1051
needs_fitting() (*evalml.pipelines.components.transformers.*
*method*), 1067
needs_fitting() (*evalml.pipelines.components.transformers.*
*method*), 1240
needs_fitting() (*evalml.pipelines.components.transformers.*
*method*), 1243
needs_fitting() (*evalml.pipelines.components.transformers.*
*method*), 1245
needs_fitting() (*evalml.pipelines.components.transformers.*
*method*), 1248
needs_fitting() (*evalml.pipelines.components.transformers.*
*method*), 1250
needs_fitting() (*evalml.pipelines.components.transformers.*
*method*), 1253
needs_fitting() (*evalml.pipelines.components.transformers.*
*method*), 1257
needs_fitting() (*evalml.pipelines.components.transformers.*
*method*), 1260
needs_fitting() (*evalml.pipelines.components.transformers.*
*method*), 1262
needs_fitting() (*evalml.pipelines.components.transformers.*
*method*), 1265
needs_fitting() (*evalml.pipelines.components.transformers.*
*method*), 1070
needs_fitting() (*evalml.pipelines.components.transformers.*
*method*), 1130
needs_fitting() (*evalml.pipelines.components.transformers.*
*method*), 1137
needs_fitting() (*evalml.pipelines.components.transformers.*
*method*), 1078
needs_fitting() (*evalml.pipelines.components.transformers.*
*method*), 1081
needs_fitting() (*evalml.pipelines.components.transformers.*
*method*), 1084
needs_fitting() (*evalml.pipelines.components.transformers.*
*method*), 1140
needs_fitting() (*evalml.pipelines.components.transformers.*
*method*), 1142
needs_fitting() (*evalml.pipelines.components.transformers.*
*method*), 1145
needs_fitting() (*evalml.pipelines.components.transformers.*
*method*), 1147
needs_fitting() (*evalml.pipelines.components.transformers.*
*method*), 1087
needs_fitting() (*evalml.pipelines.components.transformers.*
*method*), 1090
needs_fitting() (*evalml.pipelines.components.transformers.*
*method*), 1149
needs_fitting() (*evalml.pipelines.components.transformers.*
*method*), 1152
needs_fitting() (*evalml.pipelines.components.transformers.*
*method*), 1093
needs_fitting() (*evalml.pipelines.components.transformers.*
*method*), 1096
needs_fitting() (*evalml.pipelines.components.transformers.*
*method*), 1154
needs_fitting() (*evalml.pipelines.components.transformers.*
*method*), 1105
needs_fitting() (*evalml.pipelines.components.transformers.*
*method*), 1162
needs_fitting() (*evalml.pipelines.components.transformers.*
*method*), 1114
needs_fitting() (*evalml.pipelines.components.transformers.*
*method*), 1170
needs_fitting() (*evalml.pipelines.components.transformers.*
*method*), 1121
needs_fitting() (*evalml.pipelines.components.transformers.*
*method*), 1176
needs_fitting() (*evalml.pipelines.components.transformers.*

*method*), 1124
needs_fitting() (*evalml.pipelines.components.transformers.preprocessing.TimeSeriesRegularizer method*), 1127
needs_fitting() (*evalml.pipelines.components.transformers.preprocessing.TimeSeriesImputer method*), 1179
needs_fitting() (*evalml.pipelines.components.transformers.RegexTransformer method*), 1272
needs_fitting() (*evalml.pipelines.components.transformers.RFClassificationSelector method*), 1275
needs_fitting() (*evalml.pipelines.components.transformers.RFClassificationSelectFromModel method*), 1278
needs_fitting() (*evalml.pipelines.components.transformers.RFRegressionSelector method*), 1281
needs_fitting() (*evalml.pipelines.components.transformers.RFRegressionSelectFromModel method*), 1284
needs_fitting() (*evalml.pipelines.components.transformers.samplers.BaseSampler method*), 1182
needs_fitting() (*evalml.pipelines.components.transformers.samplers.OneHotSampler method*), 1191
needs_fitting() (*evalml.pipelines.components.transformers.samplers.PipelineSampler method*), 1185
needs_fitting() (*evalml.pipelines.components.transformers.samplers.SMOTENCSampler method*), 1194
needs_fitting() (*evalml.pipelines.components.transformers.samplers.SMOTENSampler method*), 1188
needs_fitting() (*evalml.pipelines.components.transformers.samplers.SMOTESampler method*), 1197
needs_fitting() (*evalml.pipelines.components.transformers.scalers.StandardScaler method*), 1200
needs_fitting() (*evalml.pipelines.components.transformers.Simplifier method*), 1291
needs_fitting() (*evalml.pipelines.components.transformers.SimpleImputer method*), 1294
needs_fitting() (*evalml.pipelines.components.transformers.TargetEncoder method*), 1302
needs_fitting() (*evalml.pipelines.components.transformers.TargetImputer method*), 1305
needs_fitting() (*evalml.pipelines.components.transformers.TimeSeriesImputer method*), 1311
needs_fitting() (*evalml.pipelines.components.transformers.TimeSeriesRegularizer method*), 1314
needs_fitting() (*evalml.pipelines.components.transformers.Transformer method*), 1317
needs_fitting() (*evalml.pipelines.components.transformers.UnivariateFeaturesTransformer method*), 1213
needs_fitting() (*evalml.pipelines.components.transformers.UnivariateImputer method*), 1320
needs_fitting() (*evalml.pipelines.components.transformers.URLFeaturizer method*), 1322
needs_fitting() (*evalml.pipelines.components.Undersampler method*), 1534
needs_fitting() (*evalml.pipelines.components.URLFeaturizer method*), 1536
needs_fitting() (*evalml.pipelines.components.VowpalWabbitBinaryClassifier method*), 1539
needs_fitting() (*evalml.pipelines.components.VowpalWabbitMulticlassClassifier method*), 1542
needs_fitting() (*evalml.pipelines.components.VowpalWabbitRegressor method*), 1545
needs_fitting() (*evalml.pipelines.components.XGBoostClassifier method*), 1549
needs_fitting() (*evalml.pipelines.components.XGBoostRegressor method*), 1551
needs_fitting() (*evalml.pipelines.DecisionTreeClassifier method*), 1673
needs_fitting() (*evalml.pipelines.DecisionTreeRegressor method*), 1677
needs_fitting() (*evalml.pipelines.DFSTransformer method*), 1680
needs_fitting() (*evalml.pipelines.DropNaNRowsTransformer method*), 1683
needs_fitting() (*evalml.pipelines.ElasticNetClassifier method*), 1686
needs_fitting() (*evalml.pipelines.ElasticNetRegressor method*), 1689
needs_fitting() (*evalml.pipelines.Estimator method*), 1692
needs_fitting() (*evalml.pipelines.ExponentialSmoothingRegressor method*), 1696
needs_fitting() (*evalml.pipelines.ExtraTreesClassifier method*), 1699
needs_fitting() (*evalml.pipelines.ExtraTreesRegressor method*), 1703
needs_fitting() (*evalml.pipelines.FeatureSelector method*), 1706
needs_fitting() (*evalml.pipelines.Imputer method*), 1709
needs_fitting() (*evalml.pipelines.KNeighborsClassifier method*), 1712
needs_fitting() (*evalml.pipelines.LightGBMClassifier method*), 1716
needs_fitting() (*evalml.pipelines.LightGBMRegressor method*), 1719
needs_fitting() (*evalml.pipelines.LinearRegressor method*), 1722
needs_fitting() (*evalml.pipelines.LogisticRegressionClassifier method*), 1725
needs_fitting() (*evalml.pipelines.OneHotEncoder method*), 1735
needs_fitting() (*evalml.pipelines.OrdinalEncoder method*), 1739
needs_fitting() (*evalml.pipelines.PerColumnImputer method*), 1741
needs_fitting() (*evalml.pipelines.ProphetRegressor method*), 1750
needs_fitting() (*evalml.pipelines.RandomForestClassifier method*), 1753
needs_fitting() (*evalml.pipelines.RandomForestRegressor*

*method*), 1756
needs_fitting() (*evalml.pipelines.RFClassifierSelectFromModel method*), 1765
needs_fitting() (*evalml.pipelines.RFRegressorSelectFromModel method*), 1768
needs_fitting() (*evalml.pipelines.SimpleImputer method*), 1771
needs_fitting() (*evalml.pipelines.StackedEnsembleBase method*), 1774
needs_fitting() (*evalml.pipelines.StackedEnsembleClassifier method*), 1778
needs_fitting() (*evalml.pipelines.StackedEnsembleRegressor method*), 1782
needs_fitting() (*evalml.pipelines.StandardScaler method*), 1784
needs_fitting() (*evalml.pipelines.SVMClassifier method*), 1787
needs_fitting() (*evalml.pipelines.SVMRegressor method*), 1790
needs_fitting() (*evalml.pipelines.TargetEncoder method*), 1793
needs_fitting() (*evalml.pipelines.TimeSeriesImputer method*), 1814
needs_fitting() (*evalml.pipelines.TimeSeriesRegularizer method*), 1833
needs_fitting() (*evalml.pipelines.Transformer method*), 1836
needs_fitting() (*evalml.pipelines.VowpalWabbitBinaryClassifier method*), 1839
needs_fitting() (*evalml.pipelines.VowpalWabbitMulticlassClassifier method*), 1842
needs_fitting() (*evalml.pipelines.VowpalWabbitRegressor method*), 1845
needs_fitting() (*evalml.pipelines.XGBoostClassifier method*), 1848
needs_fitting() (*evalml.pipelines.XGBoostRegressor method*), 1851
new() (*evalml.pipelines.binary_classification_pipeline.BinaryClassificationPipeline method*), 1558
new() (*evalml.pipelines.BinaryClassificationPipeline method*), 1650
new() (*evalml.pipelines.classification_pipeline.ClassificationPipeline method*), 1566
new() (*evalml.pipelines.ClassificationPipeline method*), 1663
new() (*evalml.pipelines.multiclass_classification_pipeline.MulticlassClassificationPipeline method*), 1579
new() (*evalml.pipelines.MulticlassClassificationPipeline method*), 1731
new() (*evalml.pipelines.pipeline_base.PipelineBase method*), 1586
new() (*evalml.pipelines.PipelineBase method*), 1746
new() (*evalml.pipelines.regression_pipeline.RegressionPipeline method*), 1594
new() (*evalml.pipelines.RegressionPipeline method*), 1762
new() (*evalml.pipelines.time_series_classification_pipelines.TimeSeriesBin*
*method*), 1601
new() (*evalml.pipelines.time_series_classification_pipelines.TimeSeriesCla*
*method*), 1609
new() (*evalml.pipelines.time_series_classification_pipelines.TimeSeriesMu*
*method*), 1616
new() (*evalml.pipelines.time_series_pipeline_base.TimeSeriesPipelineBase*
*method*), 1624
new() (*evalml.pipelines.time_series_regression_pipeline.TimeSeriesRegres*
*method*), 1633
new() (*evalml.pipelines.TimeSeriesBinaryClassificationPipeline method*), 1799
new() (*evalml.pipelines.TimeSeriesClassificationPipeline method*), 1806
new() (*evalml.pipelines.TimeSeriesMulticlassClassificationPipeline method*), 1820
new() (*evalml.pipelines.TimeSeriesRegressionPipeline method*), 1829
next_batch() (*evalml.automl.automl_algorithm.automl_algorithm.AutoM*
*method*), 224
next_batch() (*evalml.automl.automl_algorithm.AutoMLAlgorithm method*), 233
next_batch() (*evalml.automl.automl_algorithm.default_algorithm.Defaul*
*method*), 227
next_batch() (*evalml.automl.automl_algorithm.DefaultAlgorithm method*), 235
next_batch() (*evalml.automl.automl_algorithm.iterative_algorithm.Iterat*
*method*), 231
next_batch() (*evalml.automl.automl_algorithm.IterativeAlgorithm method*), 238
NoParamsException, 1891, 1893
NoPositiveLabelException, 392, 395
normalize_confusion_matrix() (*in module evalml.model_understanding*), 436
normalize_confusion_matrix() (*in module evalml.model_understanding.metrics*), 414
NoSplit (*class in evalml.preprocessing*), 1869
NoSplit (*class in evalml.preprocessing.data_splitters*), 1861
NoSplit (*class in evalml.preprocessing.data_splitters.no_split*), 1853
NoVarianceDataCheck (*class in evalml.data_checks*), 369
NoVarianceDataCheck (*class in evalml.data_checks.no_variance_data_check*), 318
NullDataCheck (*class in evalml.data_checks*), 371
NullDataCheck (*class in evalml.data_checks.null_data_check*), 321
NullsInColumnWarning, 392, 395
num_pipelines_per_batch() (*evalml.automl.automl_algorithm.automl_algorithm.AutoMLAlgo*

method), 224
num_pipelines_per_batch() (evalml.automl.automl_algorithm.AutoMLAlgorithm method), 233
num_pipelines_per_batch() (evalml.automl.automl_algorithm.default_algorithm.DefaultAlgorithm method), 227
num_pipelines_per_batch() (evalml.automl.automl_algorithm.DefaultAlgorithm method), 235
num_pipelines_per_batch() (evalml.automl.automl_algorithm.iterative_algorithm.IterativeAlgorithm method), 231
num_pipelines_per_batch() (evalml.automl.automl_algorithm.IterativeAlgorithm method), 238
number_of_features() (in module evalml.preprocessing), 1870
number_of_features() (in module evalml.preprocessing.utils), 1867
numeric_and_boolean_ww (in module evalml.utils.woodwork_utils), 1908

## O

objective_function() (evalml.objectives.AccuracyBinary method), 536
objective_function() (evalml.objectives.AccuracyMulticlass method), 537
objective_function() (evalml.objectives.AUC method), 540
objective_function() (evalml.objectives.AUCMacro method), 541
objective_function() (evalml.objectives.AUCMicro method), 543
objective_function() (evalml.objectives.AUCWeighted method), 545
objective_function() (evalml.objectives.BalancedAccuracyBinary method), 547
objective_function() (evalml.objectives.BalancedAccuracyMulticlass method), 549
objective_function() (evalml.objectives.binary_classification_objective.BinaryClassificationObjective class method), 442
objective_function() (evalml.objectives.BinaryClassificationObjective class method), 551
objective_function() (evalml.objectives.cost_benefit_matrix.CostBenefitMatrix method), 445
objective_function() (evalml.objectives.CostBenefitMatrix method), 554
objective_function() (evalml.objectives.ExpVariance method), 555
objective_function() (evalml.objectives.F1 method), 558
objective_function() (evalml.objectives.F1Macro method), 559
objective_function() (evalml.objectives.F1Micro method), 561
objective_function() (evalml.objectives.F1Weighted method), 563
objective_function() (evalml.objectives.fraud_cost.FraudCost method), 448
objective_function() (evalml.objectives.FraudCost method), 565
objective_function() (evalml.objectives.Gini method), 569
objective_function() (evalml.objectives.lead_scoring.LeadScoring method), 451
objective_function() (evalml.objectives.LeadScoring method), 571
objective_function() (evalml.objectives.LogLossBinary method), 574
objective_function() (evalml.objectives.LogLossMulticlass method), 575
objective_function() (evalml.objectives.MAE method), 577
objective_function() (evalml.objectives.MAPE method), 579
objective_function() (evalml.objectives.MaxError method), 580
objective_function() (evalml.objectives.MCCBinary method), 583
objective_function() (evalml.objectives.MCCMulticlass method), 584
objective_function() (evalml.objectives.MeanSquaredLogError method), 586
objective_function() (evalml.objectives.MedianAE method), 588
objective_function() (evalml.objectives.MSE method), 589
objective_function() (evalml.objectives.multiclass_classification_objective.MulticlassC

*class method*), 453
objective_function() (*evalml.objectives.MulticlassClassificationObjective class method*), 591
objective_function() (*evalml.objectives.objective_base.ObjectiveBase class method*), 456
objective_function() (*evalml.objectives.ObjectiveBase class method*), 594
objective_function() (*evalml.objectives.Precision method*), 596
objective_function() (*evalml.objectives.PrecisionMacro method*), 598
objective_function() (*evalml.objectives.PrecisionMicro method*), 599
objective_function() (*evalml.objectives.PrecisionWeighted method*), 601
objective_function() (*evalml.objectives.R2 method*), 603
objective_function() (*evalml.objectives.Recall method*), 605
objective_function() (*evalml.objectives.RecallMacro method*), 607
objective_function() (*evalml.objectives.RecallMicro method*), 608
objective_function() (*evalml.objectives.RecallWeighted method*), 610
objective_function() (*evalml.objectives.regression_objective.RegressionObjective class method*), 458
objective_function() (*evalml.objectives.RegressionObjective class method*), 612
objective_function() (*evalml.objectives.RootMeanSquaredError method*), 614
objective_function() (*evalml.objectives.RootMeanSquaredLogError method*), 615
objective_function() (*evalml.objectives.sensitivity_low_alert.SensitivityLowAlert method*), 461
objective_function() (*evalml.objectives.SensitivityLowAlert method*), 618
objective_function() (*evalml.objectives.standard_metrics.AccuracyBinary method*), 465
objective_function() (*evalml.objectives.standard_metrics.AccuracyMulticlass method*), 466
objective_function() (*evalml.objectives.standard_metrics.AUC method*), 469
objective_function() (*evalml.objectives.standard_metrics.AUCMacro method*), 470
objective_function() (*evalml.objectives.standard_metrics.AUCMicro method*), 472
objective_function() (*evalml.objectives.standard_metrics.AUCWeighted method*), 474
objective_function() (*evalml.objectives.standard_metrics.BalancedAccuracyBinary method*), 476
objective_function() (*evalml.objectives.standard_metrics.BalancedAccuracyMulticlass method*), 478
objective_function() (*evalml.objectives.standard_metrics.ExpVariance method*), 479
objective_function() (*evalml.objectives.standard_metrics.F1 method*), 482
objective_function() (*evalml.objectives.standard_metrics.F1Macro method*), 483
objective_function() (*evalml.objectives.standard_metrics.F1Micro method*), 485
objective_function() (*evalml.objectives.standard_metrics.F1Weighted method*), 487
objective_function() (*evalml.objectives.standard_metrics.Gini method*), 489
objective_function() (*evalml.objectives.standard_metrics.LogLossBinary method*), 491
objective_function() (*evalml.objectives.standard_metrics.LogLossMulticlass method*), 493
objective_function() (*evalml.objectives.standard_metrics.MAE method*), 494
objective_function() (*evalml.objectives.standard_metrics.MAPE method*), 496
objective_function() (*evalml.objectives.standard_metrics.MaxError

*method*), 498
objective_function() (*evalml.objectives.standard_metrics.MCCBinary method*), 500
objective_function() (*evalml.objectives.standard_metrics.MCCMulticlass method*), 502
objective_function() (*evalml.objectives.standard_metrics.MeanSquaredLogError method*), 504
objective_function() (*evalml.objectives.standard_metrics.MedianAE method*), 505
objective_function() (*evalml.objectives.standard_metrics.MSE method*), 507
objective_function() (*evalml.objectives.standard_metrics.Precision method*), 509
objective_function() (*evalml.objectives.standard_metrics.PrecisionMacro method*), 511
objective_function() (*evalml.objectives.standard_metrics.PrecisionMicro method*), 513
objective_function() (*evalml.objectives.standard_metrics.PrecisionWeighted method*), 514
objective_function() (*evalml.objectives.standard_metrics.R2 method*), 516
objective_function() (*evalml.objectives.standard_metrics.Recall method*), 518
objective_function() (*evalml.objectives.standard_metrics.RecallMacro method*), 520
objective_function() (*evalml.objectives.standard_metrics.RecallMicro method*), 522
objective_function() (*evalml.objectives.standard_metrics.RecallWeighted method*), 523
objective_function() (*evalml.objectives.standard_metrics.RootMeanSquaredError method*), 525
objective_function() (*evalml.objectives.standard_metrics.RootMeanSquaredLogError method*), 527
objective_function() (*evalml.objectives.time_series_regression_objective.TimeSeriesRegressionObjective class method*), 529
ObjectiveBase (*class in evalml.objectives*), 592
ObjectiveBase (*class in evalml.objectives.objective_base*), 454
ObjectiveCreationError, 392, 395
ObjectiveNotFoundError, 393, 395
OneHotEncoder (*class in evalml.pipelines*), 1733
OneHotEncoder (*class in evalml.pipelines.components*), 1436
OneHotEncoder (*class in evalml.pipelines.components.transformers*), 1251
OneHotEncoder (*class in evalml.pipelines.components.transformers.encoders*), 987
OneHotEncoder (*class in evalml.pipelines.components.transformers.encoders.onehot_encoder*), 971
OneHotEncoderMeta (*class in evalml.pipelines.components.transformers.encoders.onehot_encoder*), 975
optimize_threshold() (*evalml.objectives.AccuracyBinary method*), 536
optimize_threshold() (*evalml.objectives.AUC method*), 540
optimize_threshold() (*evalml.objectives.BalancedAccuracyBinary method*), 547
optimize_threshold() (*evalml.objectives.binary_classification_objective.BinaryClassificationObjective method*), 442
optimize_threshold() (*evalml.objectives.BinaryClassificationObjective method*), 551
optimize_threshold() (*evalml.objectives.cost_benefit_matrix.CostBenefitMatrix method*), 445
optimize_threshold() (*evalml.objectives.CostBenefitMatrix method*), 554
optimize_threshold() (*evalml.objectives.F1 method*), 558
optimize_threshold() (*evalml.objectives.fraud_cost.FraudCost method*), 448
optimize_threshold() (*evalml.objectives.FraudCost method*), 565
optimize_threshold() (*evalml.objectives.Gini method*), 569
optimize_threshold() (*evalml.objectives.lead_scoring.LeadScoringObjective method*), 451
optimize_threshold() (*evalml.objectives.LeadScoring method*), 571
optimize_threshold()

(*evalml.objectives.LogLossBinary method*), 574
optimize_threshold() (*evalml.objectives.MCCBinary method*), 583
optimize_threshold() (*evalml.objectives.Precision method*), 596
optimize_threshold() (*evalml.objectives.Recall method*), 605
optimize_threshold() (*evalml.objectives.sensitivity_low_alert.SensitivityLowAlert method*), 461
optimize_threshold() (*evalml.objectives.SensitivityLowAlert method*), 618
optimize_threshold() (*evalml.objectives.standard_metrics.AccuracyBinary method*), 465
optimize_threshold() (*evalml.objectives.standard_metrics.AUC method*), 469
optimize_threshold() (*evalml.objectives.standard_metrics.BalancedAccuracyBinary method*), 476
optimize_threshold() (*evalml.objectives.standard_metrics.F1 method*), 482
optimize_threshold() (*evalml.objectives.standard_metrics.Gini method*), 489
optimize_threshold() (*evalml.objectives.standard_metrics.LogLossBinary method*), 491
optimize_threshold() (*evalml.objectives.standard_metrics.MCCBinary method*), 500
optimize_threshold() (*evalml.objectives.standard_metrics.Precision method*), 509
optimize_threshold() (*evalml.objectives.standard_metrics.Recall method*), 518
optimize_threshold() (*evalml.pipelines.binary_classification_pipeline.BinaryClassificationPipeline method*), 1558
optimize_threshold() (*evalml.pipelines.binary_classification_pipeline_mixin.BinaryClassificationPipelineMixin method*), 1561
optimize_threshold() (*evalml.pipelines.BinaryClassificationPipeline method*), 1650
optimize_threshold() (*evalml.pipelines.time_series_classification_pipelines.TimeSeriesBinaryClassificationPipeline method*), 1601
optimize_threshold() (*evalml.pipelines.TimeSeriesBinaryClassificationPipeline method*), 1799
OrdinalEncoder (*class in evalml.pipelines*), 1736
OrdinalEncoder (*class in evalml.pipelines.components*), 1440
OrdinalEncoder (*class in evalml.pipelines.components.transformers*), 1254
OrdinalEncoder (*class in evalml.pipelines.components.transformers.encoders*), 990
OrdinalEncoder (*class in evalml.pipelines.components.transformers.encoders.ordinal_enco*
976
OrdinalEncoderMeta (*class in evalml.pipelines.components.transformers.encoders.ordinal_enco*
979
OutliersDataCheck (*class in evalml.data_checks*), 375
OutliersDataCheck (*class in evalml.data_checks.outliers_data_check*), 326
Oversampler (*class in evalml.pipelines.components*), 1443
Oversampler (*class in evalml.pipelines.components.transformers*), 1257
Oversampler (*class in evalml.pipelines.components.transformers.samplers*), 1189
Oversampler (*class in evalml.pipelines.components.transformers.samplers.oversampler*) 1183

## P

pad_with_nans() (*in module evalml.utils*), 1913
pad_with_nans() (*in module evalml.utils.gen_utils*), 1905
ParameterError, 1891, 1893
ParameterNotUsedWarning, 393, 396
parameters (*evalml.pipelines.ARIMARegressor property*), 1644
parameters (*evalml.pipelines.binary_classification_pipeline.BinaryClassi*
*property*), 1558
parameters (*evalml.pipelines.BinaryClassificationPipeline property*), 1651
parameters (*evalml.pipelines.CatBoostClassifier property*), 1655
parameters (*evalml.pipelines.CatBoostRegressor property*), 1658
parameters (*evalml.pipelines.classification_pipeline.ClassificationPipelin*
*property*), 1566
parameters (*evalml.pipelines.ClassificationPipeline property*), 1663

parameters (*evalml.pipelines.components.ARIMARegressor property*), 1339
parameters (*evalml.pipelines.components.BaselineClassifier property*), 1343
parameters (*evalml.pipelines.components.BaselineRegressor property*), 1346
parameters (*evalml.pipelines.components.CatBoostClassifier property*), 1349
parameters (*evalml.pipelines.components.CatBoostRegressor property*), 1352
parameters (*evalml.pipelines.components.component_base.ComponentBase property*), 1326
parameters (*evalml.pipelines.components.ComponentBase property*), 1355
parameters (*evalml.pipelines.components.DateTimeFeaturizer property*), 1358
parameters (*evalml.pipelines.components.DecisionTreeClassifier property*), 1362
parameters (*evalml.pipelines.components.DecisionTreeRegressor property*), 1366
parameters (*evalml.pipelines.components.DFSTransformer property*), 1369
parameters (*evalml.pipelines.components.DropColumns property*), 1371
parameters (*evalml.pipelines.components.DropNaNRowsTransformer property*), 1373
parameters (*evalml.pipelines.components.DropNullColumns property*), 1376
parameters (*evalml.pipelines.components.DropRowsTransformer property*), 1378
parameters (*evalml.pipelines.components.ElasticNetClassifier property*), 1381
parameters (*evalml.pipelines.components.ElasticNetRegressor property*), 1385
parameters (*evalml.pipelines.components.EmailFeaturizer property*), 1387
parameters (*evalml.pipelines.components.ensemble.StackedEnsembleBase property*), 622
parameters (*evalml.pipelines.components.ensemble.StackedEnsembleClassifier property*), 626
parameters (*evalml.pipelines.components.ensemble.StackedEnsembleRegressor property*), 630
parameters (*evalml.pipelines.components.ensemble.StackedEnsembleBase property*), 633
parameters (*evalml.pipelines.components.ensemble.StackedEnsembleClassifier property*), 637
parameters (*evalml.pipelines.components.ensemble.StackedEnsembleRegressor property*), 641
parameters (*evalml.pipelines.components.Estimator property*), 1390
parameters (*evalml.pipelines.components.estimators.ARIMARegressor property*), 861
parameters (*evalml.pipelines.components.estimators.BaselineClassifier property*), 865
parameters (*evalml.pipelines.components.estimators.BaselineRegressor property*), 868
parameters (*evalml.pipelines.components.estimators.CatBoostClassifier property*), 871
parameters (*evalml.pipelines.components.estimators.CatBoostRegressor property*), 874
parameters (*evalml.pipelines.components.estimators.classifiers.baseline_c property*), 645
parameters (*evalml.pipelines.components.estimators.classifiers.BaselineC property*), 699
parameters (*evalml.pipelines.components.estimators.classifiers.catboost_c property*), 649
parameters (*evalml.pipelines.components.estimators.classifiers.CatBoostC property*), 702
parameters (*evalml.pipelines.components.estimators.classifiers.decision_t property*), 653
parameters (*evalml.pipelines.components.estimators.classifiers.DecisionT property*), 705
parameters (*evalml.pipelines.components.estimators.classifiers.elasticnet_ property*), 657
parameters (*evalml.pipelines.components.estimators.classifiers.ElasticNet property*), 709
parameters (*evalml.pipelines.components.estimators.classifiers.et_classifi property*), 662
parameters (*evalml.pipelines.components.estimators.classifiers.ExtraTrees property*), 713
parameters (*evalml.pipelines.components.estimators.classifiers.kneighbor property*), 666
parameters (*evalml.pipelines.components.estimators.classifiers.KNeighbo property*), 716
parameters (*evalml.pipelines.components.estimators.classifiers.lightgbm_ property*), 670
parameters (*evalml.pipelines.components.estimators.classifiers.LightGBM property*), 720
parameters (*evalml.pipelines.components.estimators.classifiers.logistic_re property*), 674
parameters (*evalml.pipelines.components.estimators.classifiers.LogisticRe property*), 723
parameters (*evalml.pipelines.components.estimators.classifiers.RandomFo property*), 727
parameters (*evalml.pipelines.components.estimators.classifiers.rf_classifi property*), 677
parameters (*evalml.pipelines.components.estimators.classifiers.svm_class property*), 681
parameters (*evalml.pipelines.components.estimators.classifiers.SVMClass property*), 730
parameters (*evalml.pipelines.components.estimators.classifiers.vowpal_wa property*), 685
parameters (*evalml.pipelines.components.estimators.classifiers.vowpal_wa property*), 688
parameters (*evalml.pipelines.components.estimators.classifiers.vowpal_wa property*), 691
parameters (*evalml.pipelines.components.estimators.classifiers.VowpalWa property*), 733

parameters (evalml.pipelines.components.estimators.classifiers.VowpalWabbitMulticlassClassifier property), 737
parameters (evalml.pipelines.components.estimators.classifiers.KNearestNeighborsClassifier property), 695
parameters (evalml.pipelines.components.estimators.classifiers.XGBoostClassifier property), 740
parameters (evalml.pipelines.components.estimators.DecisionTreeClassifier property), 878
parameters (evalml.pipelines.components.estimators.DecisionTreeRegressor property), 882
parameters (evalml.pipelines.components.estimators.ElasticNetClassifier property), 886
parameters (evalml.pipelines.components.estimators.ElasticNetRegressor property), 889
parameters (evalml.pipelines.components.estimators.Estimator property), 892
parameters (evalml.pipelines.components.estimators.estimator_base.Estimator property), 856
parameters (evalml.pipelines.components.estimators.ExponentialSmoothingRegressor property), 895
parameters (evalml.pipelines.components.estimators.ExtraTreesClassifier property), 899
parameters (evalml.pipelines.components.estimators.ExtraTreesRegressor property), 903
parameters (evalml.pipelines.components.estimators.KNeighborsClassifier property), 907
parameters (evalml.pipelines.components.estimators.LightGBMClassifier property), 911
parameters (evalml.pipelines.components.estimators.LightGBMRegressor property), 914
parameters (evalml.pipelines.components.estimators.LinearRegressor property), 917
parameters (evalml.pipelines.components.estimators.LogisticRegressionClassifier property), 921
parameters (evalml.pipelines.components.estimators.ProphetRegressor property), 924
parameters (evalml.pipelines.components.estimators.RandomForestClassifier property), 927
parameters (evalml.pipelines.components.estimators.RandomForestRegressor property), 930
parameters (evalml.pipelines.components.estimators.regressors.ARIMARegressor property), 745
parameters (evalml.pipelines.components.estimators.regressors.ARIMARegressor property), 804
parameters (evalml.pipelines.components.estimators.regressors.BaselineRegressor property), 749
parameters (evalml.pipelines.components.estimators.regressors.CatBoostRegressor property), 808
parameters (evalml.pipelines.components.estimators.regressors.CatBoostRegressor property), 752
parameters (evalml.pipelines.components.estimators.regressors.DecisionTreeRegressor property), 811
parameters (evalml.pipelines.components.estimators.regressors.DecisionTreeRegressor property), 757
parameters (evalml.pipelines.components.estimators.regressors.DecisionTreeRegressor property), 815
parameters (evalml.pipelines.components.estimators.regressors.elasticnet property), 760
parameters (evalml.pipelines.components.estimators.regressors.ElasticNet property), 818
parameters (evalml.pipelines.components.estimators.regressors.et_regressor property), 765
parameters (evalml.pipelines.components.estimators.regressors.exponential property), 769
parameters (evalml.pipelines.components.estimators.regressors.Exponential property), 822
parameters (evalml.pipelines.components.estimators.regressors.ExtraTreesRegressor property), 825
parameters (evalml.pipelines.components.estimators.regressors.lightgbm_regressor property), 773
parameters (evalml.pipelines.components.estimators.regressors.LightGBMRegressor property), 829
parameters (evalml.pipelines.components.estimators.regressors.linear_regressor property), 776
parameters (evalml.pipelines.components.estimators.regressors.LinearRegressor property), 832
parameters (evalml.pipelines.components.estimators.regressors.prophet_regressor property), 781
parameters (evalml.pipelines.components.estimators.regressors.ProphetRegressor property), 836
parameters (evalml.pipelines.components.estimators.regressors.RandomForestRegressor property), 839
parameters (evalml.pipelines.components.estimators.regressors.rf_regressor property), 785
parameters (evalml.pipelines.components.estimators.regressors.svm_regressor property), 789
parameters (evalml.pipelines.components.estimators.regressors.SVMRegressor property), 843
parameters (evalml.pipelines.components.estimators.regressors.time_series property), 792
parameters (evalml.pipelines.components.estimators.regressors.TimeSeries property), 846
parameters (evalml.pipelines.components.estimators.regressors.vowpal_wabbit_regressor property), 796
parameters (evalml.pipelines.components.estimators.regressors.VowpalWabbitBinaryRegressor property), 849
parameters (evalml.pipelines.components.estimators.regressors.xgboost_regressor property), 799
parameters (evalml.pipelines.components.estimators.regressors.XGBoostRegressor property), 852
parameters (evalml.pipelines.components.estimators.SVMClassifier property), 934
parameters (evalml.pipelines.components.estimators.SVMRegressor property), 937
parameters (evalml.pipelines.components.estimators.TimeSeriesBaselineEstimator property), 940
parameters (evalml.pipelines.components.VowpalWabbitBinaryRegressor property), 943

parameters (*evalml.pipelines.components.estimators.VowpalWabbitMulticlassClassifier property*), 947
parameters (*evalml.pipelines.components.estimators.VowpalWabbitRegressor property*), 950
parameters (*evalml.pipelines.components.estimators.XGBoostClassifier property*), 953
parameters (*evalml.pipelines.components.estimators.XGBoostRegressor property*), 956
parameters (*evalml.pipelines.components.ExponentialSmoothingRegressor property*), 1393
parameters (*evalml.pipelines.components.ExtraTreesClassifier property*), 1397
parameters (*evalml.pipelines.components.ExtraTreesRegressor property*), 1401
parameters (*evalml.pipelines.components.FeatureSelector property*), 1404
parameters (*evalml.pipelines.components.Imputer property*), 1406
parameters (*evalml.pipelines.components.KNeighborsClassifier property*), 1410
parameters (*evalml.pipelines.components.LabelEncoder property*), 1412
parameters (*evalml.pipelines.components.LightGBMClassifier property*), 1416
parameters (*evalml.pipelines.components.LightGBMRegressor property*), 1419
parameters (*evalml.pipelines.components.LinearDiscriminantAnalysis property*), 1422
parameters (*evalml.pipelines.components.LinearRegressor property*), 1425
parameters (*evalml.pipelines.components.LogisticRegressionClassifier property*), 1428
parameters (*evalml.pipelines.components.LogTransformer property*), 1431
parameters (*evalml.pipelines.components.LSA property*), 1433
parameters (*evalml.pipelines.components.NaturalLanguageFeaturizer property*), 1436
parameters (*evalml.pipelines.components.OneHotEncoder property*), 1439
parameters (*evalml.pipelines.components.OrdinalEncoder property*), 1443
parameters (*evalml.pipelines.components.Oversampler property*), 1445
parameters (*evalml.pipelines.components.PCA property*), 1448
parameters (*evalml.pipelines.components.PerColumnImputer property*), 1450
parameters (*evalml.pipelines.components.PolynomialDecomposer property*), 1455
parameters (*evalml.pipelines.components.ProphetRegressor property*), 1458
parameters (*evalml.pipelines.components.RandomForestClassifier property*), 1462
parameters (*evalml.pipelines.components.RandomForestRegressor property*), 1465
parameters (*evalml.pipelines.components.ReplaceNullableTypes property*), 1467
parameters (*evalml.pipelines.components.RFClassifierRFESelector property*), 1470
parameters (*evalml.pipelines.components.RFClassifierSelectFromModel property*), 1473
parameters (*evalml.pipelines.components.RFRegressorRFESelector property*), 1476
parameters (*evalml.pipelines.components.RFRegressorSelectFromModel property*), 1479
parameters (*evalml.pipelines.components.SelectByType property*), 1481
parameters (*evalml.pipelines.components.SelectColumns property*), 1484
parameters (*evalml.pipelines.components.SimpleImputer property*), 1486
parameters (*evalml.pipelines.components.StackedEnsembleBase property*), 1489
parameters (*evalml.pipelines.components.StackedEnsembleClassifier property*), 1493
parameters (*evalml.pipelines.components.StackedEnsembleRegressor property*), 1497
parameters (*evalml.pipelines.components.StandardScaler property*), 1499
parameters (*evalml.pipelines.components.STLDecomposer property*), 1504
parameters (*evalml.pipelines.components.SVMClassifier property*), 1508
parameters (*evalml.pipelines.components.SVMRegressor property*), 1511
parameters (*evalml.pipelines.components.TargetEncoder property*), 1514
parameters (*evalml.pipelines.components.TargetImputer property*), 1516
parameters (*evalml.pipelines.components.TimeSeriesBaselineEstimator property*), 1519
parameters (*evalml.pipelines.components.TimeSeriesFeaturizer property*), 1522
parameters (*evalml.pipelines.components.TimeSeriesImputer property*), 1525
parameters (*evalml.pipelines.components.TimeSeriesRegularizer property*), 1528
parameters (*evalml.pipelines.components.Transformer property*), 1531
parameters (*evalml.pipelines.components.transformers.column_selectors.* property), 1203
parameters (*evalml.pipelines.components.transformers.column_selectors.* property), 1205
parameters (*evalml.pipelines.components.transformers.column_selectors.* property), 1208
parameters (*evalml.pipelines.components.transformers.column_selectors.* property), 1210

parameters (evalml.pipelines.components.transformers.DateFeatures property), 1217
parameters (evalml.pipelines.components.transformers.DFSTransformer property), 1220
parameters (evalml.pipelines.components.transformers.dimensionality_reduction.LinearDiscriminantAnalysis property), 959
parameters (evalml.pipelines.components.transformers.dimensionality_reduction.LinearDiscriminantAnalysis property), 965
parameters (evalml.pipelines.components.transformers.dimensionality_reduction.PCA property), 967
parameters (evalml.pipelines.components.transformers.dimensionality_reduction.PCA property), 962
parameters (evalml.pipelines.components.transformers.DropColumns property), 1223
parameters (evalml.pipelines.components.transformers.DropNullColumns property), 1225
parameters (evalml.pipelines.components.transformers.DropRowsTransformer property), 1227
parameters (evalml.pipelines.components.transformers.EmptyFeaturizer property), 1230
parameters (evalml.pipelines.components.transformers.encoders.CatBoostEncoder property), 970
parameters (evalml.pipelines.components.transformers.encoders.EstimatorTargetEncoder property), 986
parameters (evalml.pipelines.components.transformers.encoders.JSONTargetEncoder property), 974
parameters (evalml.pipelines.components.transformers.encoders.LeaveOneOutEncoder property), 989
parameters (evalml.pipelines.components.transformers.encoders.OneHotEncoder property), 979
parameters (evalml.pipelines.components.transformers.encoders.OrdinalEncoder property), 993
parameters (evalml.pipelines.components.transformers.encoders.TargetEncoder property), 983
parameters (evalml.pipelines.components.transformers.encoders.TimeSeriesEncoder property), 995
parameters (evalml.pipelines.components.transformers.feature_selection.ExpandedFeatureSelector property), 999
parameters (evalml.pipelines.components.transformers.feature_selection.FeatureSelector property), 1019
parameters (evalml.pipelines.components.transformers.feature_selection.RecursiveFeatureElimination property), 1002
parameters (evalml.pipelines.components.transformers.feature_selection.RFCheckpointer property), 1005
parameters (evalml.pipelines.components.transformers.feature_selection.RFClassifierSelector property), 1008
parameters (evalml.pipelines.components.transformers.feature_selection.RFRegressorSelector property), 1012
parameters (evalml.pipelines.components.transformers.feature_selection.SelectFromModel property), 1015
parameters (evalml.pipelines.components.transformers.feature_selection.RFECVSelector property), 1022
parameters (evalml.pipelines.components.transformers.feature_selection.RFECVSelector property), 1025
parameters (evalml.pipelines.components.transformers.feature_selection.RFECVSelector property), 1028
parameters (evalml.pipelines.components.transformers.feature_selection.RFECVSelector property), 1031
parameters (evalml.pipelines.components.transformers.FeatureSelector property), 1235
parameters (evalml.pipelines.components.transformers.Imputer property), 1238
parameters (evalml.pipelines.components.transformers.imputers.Imputer property), 1054
parameters (evalml.pipelines.components.transformers.imputers.imputer.I property), 1035
parameters (evalml.pipelines.components.transformers.imputers.knn_impu property), 1038
parameters (evalml.pipelines.components.transformers.imputers.KNNImp property), 1056
parameters (evalml.pipelines.components.transformers.imputers.per_colu property), 1041
parameters (evalml.pipelines.components.transformers.imputers.PerColum property), 1059
parameters (evalml.pipelines.components.transformers.imputers.simple_im property), 1044
parameters (evalml.pipelines.components.transformers.imputers.SimpleIm property), 1061
parameters (evalml.pipelines.components.transformers.imputers.target_im property), 1047
parameters (evalml.pipelines.components.transformers.imputers.TargetImp property), 1064
parameters (evalml.pipelines.components.transformers.imputers.time_seri property), 1051
parameters (evalml.pipelines.components.transformers.imputers.TimeSeri property), 1067
parameters (evalml.pipelines.components.transformers.LabelEncoder property), 1240
parameters (evalml.pipelines.components.transformers.LinearDiscriminan property), 1243
parameters (evalml.pipelines.components.transformers.LogTransformer property), 1245
parameters (evalml.pipelines.components.transformers.LSA property), 1248
parameters (evalml.pipelines.components.transformers.RecursiveFeatureEli property), 1250
parameters (evalml.pipelines.components.transformers.RFCheckpointer property), 1254
parameters (evalml.pipelines.components.transformers.RFClassifierSelec property), 1257
parameters (evalml.pipelines.components.transformers.RFRegressorSelect property), 1260
parameters (evalml.pipelines.components.transformers.SelectFromModel property), 1262
parameters (evalml.pipelines.components.transformers.PerColumnImpute property), 1265

parameters (evalml.pipelines.components.transformers.Po
property), 1270
parameters (evalml.pipelines.components.transformers.pr
property), 1070
parameters (evalml.pipelines.components.transformers.pr
property), 1130
parameters (evalml.pipelines.components.transformers.pr
property), 1134
parameters (evalml.pipelines.components.transformers.pr
property), 1075
parameters (evalml.pipelines.components.transformers.pr
property), 1137
parameters (evalml.pipelines.components.transformers.pr
property), 1078
parameters (evalml.pipelines.components.transformers.pr
property), 1081
parameters (evalml.pipelines.components.transformers.pr
property), 1084
parameters (evalml.pipelines.components.transformers.pr
property), 1140
parameters (evalml.pipelines.components.transformers.pr
property), 1142
parameters (evalml.pipelines.components.transformers.pr
property), 1145
parameters (evalml.pipelines.components.transformers.pr
property), 1147
parameters (evalml.pipelines.components.transformers.pr
property), 1087
parameters (evalml.pipelines.components.transformers.pr
property), 1090
parameters (evalml.pipelines.components.transformers.pr
property), 1150
parameters (evalml.pipelines.components.transformers.pr
property), 1152
parameters (evalml.pipelines.components.transformers.pr
property), 1093
parameters (evalml.pipelines.components.transformers.pr
property), 1096
parameters (evalml.pipelines.components.transformers.pr
property), 1154
parameters (evalml.pipelines.components.transformers.pr
property), 1102
parameters (evalml.pipelines.components.transformers.pr
property), 1159
parameters (evalml.pipelines.components.transformers.pr
property), 1105
parameters (evalml.pipelines.components.transformers.pr
property), 1162
parameters (evalml.pipelines.components.transformers.pr
property), 1111
parameters (evalml.pipelines.components.transformers.pr
property), 1167
parameters (evalml.pipelines.components.transformers.pr
property), 1114
parameters (evalml.pipelines.components.transformers.preprocessing.Tex
property), 1170
parameters (evalml.pipelines.components.transformers.preprocessing.time
property), 1118
parameters (evalml.pipelines.components.transformers.preprocessing.time
property), 1121
parameters (evalml.pipelines.components.transformers.preprocessing.Tim
property), 1173
parameters (evalml.pipelines.components.transformers.preprocessing.Tim
property), 1176
parameters (evalml.pipelines.components.transformers.preprocessing.tran
property), 1124
parameters (evalml.pipelines.components.transformers.preprocessing.tran
property), 1127
parameters (evalml.pipelines.components.transformers.preprocessing.URL
property), 1179
parameters (evalml.pipelines.components.transformers.ReplaceNullableTy
property), 1272
parameters (evalml.pipelines.components.transformers.RFClassifierRFES
property), 1275
parameters (evalml.pipelines.components.transformers.RFClassifierSelect
property), 1278
parameters (evalml.pipelines.components.transformers.RFRegressorRFES
property), 1281
parameters (evalml.pipelines.components.transformers.RFRegressorSelec
property), 1284
parameters (evalml.pipelines.components.transformers.samplers.base_sam
property), 1182
parameters (evalml.pipelines.components.transformers.samplers.Oversam
property), 1191
parameters (evalml.pipelines.components.transformers.samplers.oversamp
property), 1185
parameters (evalml.pipelines.components.transformers.samplers.Undersa
property), 1194
parameters (evalml.pipelines.components.transformers.samplers.undersa
property), 1188
parameters (evalml.pipelines.components.transformers.scalers.StandardS
property), 1200
parameters (evalml.pipelines.components.transformers.SelectByType
property), 1287
parameters (evalml.pipelines.components.transformers.SelectColumns
property), 1289
parameters (evalml.pipelines.components.transformers.SimpleImputer
property), 1291
parameters (evalml.pipelines.components.transformers.StandardScaler
property), 1294
parameters (evalml.pipelines.components.transformers.STLDecomposer
property), 1299
parameters (evalml.pipelines.components.transformers.TargetEncoder
property), 1302
parameters (evalml.pipelines.components.transformers.TargetImputer
property), 1305

parameters (*evalml.pipelines.components.transformers.TimeSeriesFeaturizer property*), 1308
parameters (*evalml.pipelines.components.transformers.TimeSeriesImputer property*), 1311
parameters (*evalml.pipelines.components.transformers.TimeSeriesRegularizer property*), 1314
parameters (*evalml.pipelines.components.transformers.Transformer property*), 1317
parameters (*evalml.pipelines.components.transformers.transformer.Transformer property*), 1213
parameters (*evalml.pipelines.components.transformers.Unstacker property*), 1320
parameters (*evalml.pipelines.components.transformers.URLFeaturizer property*), 1323
parameters (*evalml.pipelines.components.Undersampler property*), 1534
parameters (*evalml.pipelines.components.URLFeaturizer property*), 1536
parameters (*evalml.pipelines.components.VowpalWabbitBinaryClassifier property*), 1540
parameters (*evalml.pipelines.components.VowpalWabbitMulticlassClassifier property*), 1543
parameters (*evalml.pipelines.components.VowpalWabbitRegressor property*), 1546
parameters (*evalml.pipelines.components.XGBoostClassifier property*), 1549
parameters (*evalml.pipelines.components.XGBoostRegressor property*), 1552
parameters (*evalml.pipelines.DecisionTreeClassifier property*), 1674
parameters (*evalml.pipelines.DecisionTreeRegressor property*), 1677
parameters (*evalml.pipelines.DFSTransformer property*), 1680
parameters (*evalml.pipelines.DropNaNRowsTransformer property*), 1683
parameters (*evalml.pipelines.ElasticNetClassifier property*), 1686
parameters (*evalml.pipelines.ElasticNetRegressor property*), 1689
parameters (*evalml.pipelines.Estimator property*), 1692
parameters (*evalml.pipelines.ExponentialSmoothingRegressor property*), 1696
parameters (*evalml.pipelines.ExtraTreesClassifier property*), 1700
parameters (*evalml.pipelines.ExtraTreesRegressor property*), 1703
parameters (*evalml.pipelines.FeatureSelector property*), 1706
parameters (*evalml.pipelines.Imputer property*), 1709
parameters (*evalml.pipelines.KNeighborsClassifier property*), 1712
parameters (*evalml.pipelines.LightGBMClassifier property*), 1716
parameters (*evalml.pipelines.LightGBMRegressor property*), 1719
parameters (*evalml.pipelines.LinearRegressor property*), 1722
parameters (*evalml.pipelines.LogisticRegressionClassifier property*), 1726
parameters (*evalml.pipelines.multiclass_classification_pipeline.MulticlassClassificationPipeline property*), 1579
parameters (*evalml.pipelines.MulticlassClassificationPipeline property*), 1731
parameters (*evalml.pipelines.OneHotEncoder property*), 1736
parameters (*evalml.pipelines.OrdinalEncoder property*), 1739
parameters (*evalml.pipelines.PerColumnImputer property*), 1741
parameters (*evalml.pipelines.pipeline_base.PipelineBase property*), 1586
parameters (*evalml.pipelines.PipelineBase property*), 1746
parameters (*evalml.pipelines.ProphetRegressor property*), 1750
parameters (*evalml.pipelines.RandomForestClassifier property*), 1753
parameters (*evalml.pipelines.RandomForestRegressor property*), 1756
parameters (*evalml.pipelines.regression_pipeline.RegressionPipeline property*), 1594
parameters (*evalml.pipelines.RegressionPipeline property*), 1762
parameters (*evalml.pipelines.RFClassifierSelectFromModel property*), 1766
parameters (*evalml.pipelines.RFRegressorSelectFromModel property*), 1768
parameters (*evalml.pipelines.SimpleImputer property*), 1771
parameters (*evalml.pipelines.StackedEnsembleBase property*), 1774
parameters (*evalml.pipelines.StackedEnsembleClassifier property*), 1778
parameters (*evalml.pipelines.StackedEnsembleRegressor property*), 1782
parameters (*evalml.pipelines.StandardScaler property*), 1784
parameters (*evalml.pipelines.SVMClassifier property*), 1787
parameters (*evalml.pipelines.SVMRegressor property*), 1791
parameters (*evalml.pipelines.TargetEncoder property*), 1793
parameters (*evalml.pipelines.time_series_classification_pipelines.TimeSeriesClassificationPipeline property*), 1602
parameters (*evalml.pipelines.time_series_classification_pipelines.TimeSeriesMulticlassClassificationPipeline property*), 1609

parameters (*evalml.pipelines.time_series_classification_pipelines.TimeSeriesMulticlassClassificationPipeline property*), 1617
parameters (*evalml.pipelines.time_series_pipeline_base.TimeSeriesPipelineBase property*), 1624
parameters (*evalml.pipelines.time_series_regression_pipeline_base.TimeSeriesRegressionPipeline property*), 1633
parameters (*evalml.pipelines.TimeSeriesBinaryClassificationPipeline property*), 1800
parameters (*evalml.pipelines.TimeSeriesClassificationPipeline property*), 1807
parameters (*evalml.pipelines.TimeSeriesFeaturizer property*), 1811
parameters (*evalml.pipelines.TimeSeriesImputer property*), 1814
parameters (*evalml.pipelines.TimeSeriesMulticlassClassificationPipeline property*), 1820
parameters (*evalml.pipelines.TimeSeriesRegressionPipeline property*), 1829
parameters (*evalml.pipelines.TimeSeriesRegularizer property*), 1833
parameters (*evalml.pipelines.Transformer property*), 1836
parameters (*evalml.pipelines.VowpalWabbitBinaryClassifier property*), 1839
parameters (*evalml.pipelines.VowpalWabbitMulticlassClassifier property*), 1842
parameters (*evalml.pipelines.VowpalWabbitRegressor property*), 1845
parameters (*evalml.pipelines.XGBoostClassifier property*), 1849
parameters (*evalml.pipelines.XGBoostRegressor property*), 1851
partial_dependence() (*in module evalml.model_understanding*), 437
partial_dependence() (*in module evalml.model_understanding.partial_dependence_plots*), 417
PartialDependenceError, 393, 396
PartialDependenceErrorCode (*class in evalml.exceptions*), 396
PartialDependenceErrorCode (*class in evalml.exceptions.exceptions*), 393
PCA (*class in evalml.pipelines.components*), 1446
PCA (*class in evalml.pipelines.components.transformers*), 1260
PCA (*class in evalml.pipelines.components.transformers.dimensionality_reduction*), 965
PCA (*class in evalml.pipelines.components.transformers.dimensionality_reduction*), 960
PerColumnImputer (*class in evalml.pipelines*), 1739
PerColumnImputer (*class in evalml.pipelines.components*), 1448
PerColumnImputer (*class in evalml.pipelines.components.transformers*), 1263
PerColumnImputer (*class in evalml.pipelines.components.transformers.imputers*), 1057
PerColumnImputer (*class in evalml.pipelines.components.transformers.imputers.per_column_imputer*), 1039
perfect_score (*evalml.objectives.binary_classification_objective.BinaryClassificationObjective property*), 442
perfect_score (*evalml.objectives.BinaryClassificationObjective property*), 551
perfect_score (*evalml.objectives.multiclass_classification_objective.MulticlassClassificationObjective property*), 453
perfect_score (*evalml.objectives.MulticlassClassificationObjective property*), 591
perfect_score (*evalml.objectives.objective_base.ObjectiveBase property*), 456
perfect_score (*evalml.objectives.ObjectiveBase property*), 594
perfect_score (*evalml.objectives.regression_objective.RegressionObjective property*), 458
perfect_score (*evalml.objectives.RegressionObjective property*), 612
perfect_score (*evalml.objectives.time_series_regression_objective.TimeSeriesRegressionObjective property*), 529
pipeline_number (*evalml.automl.automl_algorithm.automl_algorithm.AutoMLAlgorithm property*), 225
pipeline_number (*evalml.automl.automl_algorithm.AutoMLAlgorithm property*), 233
pipeline_number (*evalml.automl.automl_algorithm.default_algorithm.DefaultAlgorithm property*), 228
pipeline_number (*evalml.automl.automl_algorithm.DefaultAlgorithm property*), 236
pipeline_number (*evalml.automl.automl_algorithm.iterative_algorithm.IterativeAlgorithm property*), 231
pipeline_number (*evalml.automl.automl_algorithm.IterativeAlgorithm property*), 238
PipelineBase (*class in evalml.pipelines*), 1742
PipelineBase (*class in evalml.pipelines.pipeline_base*), 1581
PipelineBaseMeta (*class in evalml.pipelines.pipeline_meta*), 1588
PipelineError, 393, 396
PipelineErrorCodeEnum (*class in evalml.exceptions*), 397
PipelineErrorCodeEnum (*class in evalml.exceptions.exceptions*), 394
PipelineNotFoundError, 394, 397
PipelineNotYetFittedError, 394, 397
PipelineScoreError, 394, 397
PipelineSearchPlots (*class in evalml.automl.pipeline_search_plots*), 267
plot (*evalml.automl.automl_search.AutoMLSearch property*), 262

plot (*evalml.automl.AutoMLSearch property*), 278
plot (*evalml.AutoMLSearch property*), 1918
plot_decomposition() (*evalml.pipelines.components.PolynomialDecomposer method*), 1455
plot_decomposition() (*evalml.pipelines.components.STLDecomposer method*), 1504
plot_decomposition() (*evalml.pipelines.components.transformers.PolynomialDecomposer method*), 1270
plot_decomposition() (*evalml.pipelines.components.transformers.preprocessing.DelayedFeatureDecomposer method*), 1134
plot_decomposition() (*evalml.pipelines.components.transformers.preprocessing.DelayedFeatureDecomposer method*), 1075
plot_decomposition() (*evalml.pipelines.components.transformers.preprocessing.polynomial_decomposer.PolynomialDecomposer method*), 1102
plot_decomposition() (*evalml.pipelines.components.transformers.preprocessing.PolynomialDecomposer method*), 1159
plot_decomposition() (*evalml.pipelines.components.transformers.preprocessing.stl_decomposer.STLDecomposer method*), 1111
plot_decomposition() (*evalml.pipelines.components.transformers.preprocessing.STLDecomposer method*), 1167
plot_decomposition() (*evalml.pipelines.components.transformers.STLDecomposer method*), 1299
PolynomialDecomposer (*class in evalml.pipelines.components*), 1451
PolynomialDecomposer (*class in evalml.pipelines.components.transformers*), 1265
PolynomialDecomposer (*class in evalml.pipelines.components.transformers.preprocessing*), 1155
PolynomialDecomposer (*class in evalml.pipelines.components.transformers.preprocessing.polynomial_decomposer*), 1097
positive_only() (*evalml.objectives.AccuracyBinary method*), 536
positive_only() (*evalml.objectives.AccuracyMulticlass method*), 537
positive_only() (*evalml.objectives.AUC method*), 540
positive_only() (*evalml.objectives.AUCMacro method*), 541
positive_only() (*evalml.objectives.AUCMicro method*), 543
positive_only() (*evalml.objectives.AUCWeighted method*), 545
positive_only() (*evalml.objectives.BalancedAccuracyBinary method*), 547
positive_only() (*evalml.objectives.BalancedAccuracyMulticlass method*), 549
positive_only() (*evalml.objectives.binary_classification_objective.BinaryClassificationObjective method*), 443
positive_only() (*evalml.objectives.BinaryClassificationObjective method*), 552
positive_only() (*evalml.objectives.cost_benefit_matrix.CostBenefitMatrix method*), 446
positive_only() (*evalml.objectives.CostBenefitMatrix method*), 554
positive_only() (*evalml.objectives.ExpVariance method*), 556
positive_only() (*evalml.objectives.F1 method*), 558
positive_only() (*evalml.objectives.F1Macro method*), 559
positive_only() (*evalml.objectives.F1Micro method*), 561
positive_only() (*evalml.objectives.F1Weighted method*), 563
positive_only() (*evalml.objectives.fraud_cost.FraudCost method*), 448
positive_only() (*evalml.objectives.FraudCost method*), 565
positive_only() (*evalml.objectives.Gini method*), 569
positive_only() (*evalml.objectives.lead_scoring.LeadScoring method*), 451
positive_only() (*evalml.objectives.LeadScoring method*), 571
positive_only() (*evalml.objectives.LogLossBinary method*), 574
positive_only() (*evalml.objectives.LogLossMulticlass method*), 575
positive_only() (*evalml.objectives.MAE method*), 577
positive_only() (*evalml.objectives.MAPE method*), 579
positive_only() (*evalml.objectives.MaxError method*), 581
positive_only() (*evalml.objectives.MCCBinary method*), 583
positive_only() (*evalml.objectives.MCCMulticlass method*), 584
positive_only() (*evalml.objectives.MeanSquaredLogError method*), 586
positive_only() (*evalml.objectives.MedianAE method*), 588
positive_only() (*evalml.objectives.MSE method*), 589
positive_only() (*evalml.objectives.multiclass_classification_objective.MulticlassClassificationObjective method*), 453
positive_only() (*evalml.objectives.MulticlassClassificationObjective method*), 591
positive_only() (*evalml.objectives.objective_base.ObjectiveBase method*), 456

positive_only() (*evalml.objectives.ObjectiveBase method*), 594
positive_only() (*evalml.objectives.Precision method*), 596
positive_only() (*evalml.objectives.PrecisionMacro method*), 598
positive_only() (*evalml.objectives.PrecisionMicro method*), 599
positive_only() (*evalml.objectives.PrecisionWeighted method*), 601
positive_only() (*evalml.objectives.R2 method*), 603
positive_only() (*evalml.objectives.Recall method*), 605
positive_only() (*evalml.objectives.RecallMacro method*), 607
positive_only() (*evalml.objectives.RecallMicro method*), 608
positive_only() (*evalml.objectives.RecallWeighted method*), 610
positive_only() (*evalml.objectives.regression_objective.RegressionObjective method*), 458
positive_only() (*evalml.objectives.RegressionObjective method*), 612
positive_only() (*evalml.objectives.RootMeanSquaredError method*), 614
positive_only() (*evalml.objectives.RootMeanSquaredLogError method*), 616
positive_only() (*evalml.objectives.sensitivity_low_alert.SensitivityLowAlert method*), 461
positive_only() (*evalml.objectives.SensitivityLowAlert method*), 618
positive_only() (*evalml.objectives.standard_metrics.AccuracyBinary method*), 465
positive_only() (*evalml.objectives.standard_metrics.AccuracyMulticlass method*), 466
positive_only() (*evalml.objectives.standard_metrics.AUC method*), 469
positive_only() (*evalml.objectives.standard_metrics.AUCMacro method*), 470
positive_only() (*evalml.objectives.standard_metrics.AUCMicro method*), 472
positive_only() (*evalml.objectives.standard_metrics.AUCWeighted method*), 474
positive_only() (*evalml.objectives.standard_metrics.BalancedAccuracyBinary method*), 476
positive_only() (*evalml.objectives.standard_metrics.BalancedAccuracyMulticlass method*), 478
positive_only() (*evalml.objectives.standard_metrics.ExplainedVariance method*), 479
positive_only() (*evalml.objectives.standard_metrics.F1 method*), 482
positive_only() (*evalml.objectives.standard_metrics.F1Macro method*), 483
positive_only() (*evalml.objectives.standard_metrics.F1Micro method*), 485
positive_only() (*evalml.objectives.standard_metrics.F1Weighted method*), 487
positive_only() (*evalml.objectives.standard_metrics.Gini method*), 489
positive_only() (*evalml.objectives.standard_metrics.LogLossBinary method*), 491
positive_only() (*evalml.objectives.standard_metrics.LogLossMulticlass method*), 493
positive_only() (*evalml.objectives.standard_metrics.MAE method*), 495
positive_only() (*evalml.objectives.standard_metrics.MAPE method*), 496
positive_only() (*evalml.objectives.standard_metrics.MaxError method*), 498
positive_only() (*evalml.objectives.standard_metrics.MCCBinary method*), 500
positive_only() (*evalml.objectives.standard_metrics.MCCMulticlass method*), 502
positive_only() (*evalml.objectives.standard_metrics.MeanSquaredLogE method*), 504
positive_only() (*evalml.objectives.standard_metrics.MedianAE method*), 506
positive_only() (*evalml.objectives.standard_metrics.MSE method*), 507
positive_only() (*evalml.objectives.standard_metrics.Precision method*), 509
positive_only() (*evalml.objectives.standard_metrics.PrecisionMacro method*), 511
positive_only() (*evalml.objectives.standard_metrics.PrecisionMicro method*), 513
positive_only() (*evalml.objectives.standard_metrics.PrecisionWeighted method*), 514
positive_only() (*evalml.objectives.standard_metrics.R2 method*), 516
positive_only() (*evalml.objectives.standard_metrics.Recall method*), 518
positive_only() (*evalml.objectives.standard_metrics.RecallMacro method*), 520
positive_only() (*evalml.objectives.standard_metrics.RecallMicro method*), 522
positive_only() (*evalml.objectives.standard_metrics.RecallWeighted method*), 523
positive_only() (*evalml.objectives.standard_metrics.RootMeanSquared method*), 525
positive_only() (*evalml.objectives.standard_metrics.RootMeanSquared method*), 527
positive_only() (*evalml.objectives.time_series_regression_objective.Tim method*), 529
Precision (*class in evalml.objectives*), 594
Precision (*class in evalml.objectives.standard_metrics*), 508
precision_recall_curve() (*in module evalml.model_understanding*), 438

precision_recall_curve() (*in module evalml.model_understanding.metrics*), 415
PrecisionMacro (*class in evalml.objectives*), 597
PrecisionMacro (*class in evalml.objectives.standard_metrics*), 510
PrecisionMicro (*class in evalml.objectives*), 598
PrecisionMicro (*class in evalml.objectives.standard_metrics*), 512
PrecisionWeighted (*class in evalml.objectives*), 600
PrecisionWeighted (*class in evalml.objectives.standard_metrics*), 513
predict() (*evalml.pipelines.ARIMARegressor method*), 1644
predict() (*evalml.pipelines.binary_classification_pipeline.BinaryClassificationPipeline method*), 1559
predict() (*evalml.pipelines.BinaryClassificationPipeline method*), 1651
predict() (*evalml.pipelines.CatBoostClassifier method*), 1655
predict() (*evalml.pipelines.CatBoostRegressor method*), 1658
predict() (*evalml.pipelines.classification_pipeline.ClassificationPipeline method*), 1566
predict() (*evalml.pipelines.ClassificationPipeline method*), 1663
predict() (*evalml.pipelines.component_graph.ComponentGraph method*), 1573
predict() (*evalml.pipelines.ComponentGraph method*), 1670
predict() (*evalml.pipelines.components.ARIMARegressor method*), 1339
predict() (*evalml.pipelines.components.BaselineClassifier method*), 1343
predict() (*evalml.pipelines.components.BaselineRegressor method*), 1346
predict() (*evalml.pipelines.components.CatBoostClassifier method*), 1349
predict() (*evalml.pipelines.components.CatBoostRegressor method*), 1353
predict() (*evalml.pipelines.components.DecisionTreeClassifier method*), 1362
predict() (*evalml.pipelines.components.DecisionTreeRegressor method*), 1366
predict() (*evalml.pipelines.components.ElasticNetClassifier method*), 1382
predict() (*evalml.pipelines.components.ElasticNetRegressor method*), 1385
predict() (*evalml.pipelines.components.ensemble.StackedEnsembleBase method*), 622
predict() (*evalml.pipelines.components.ensemble.StackedEnsembleClassifier method*), 626
predict() (*evalml.pipelines.components.ensemble.StackedEnsembleRegressor method*), 630
predict() (*evalml.pipelines.components.ensemble.Stacked... method*), 633
predict() (*evalml.pipelines.components.ensemble.StackedEnsembleClass... method*), 637
predict() (*evalml.pipelines.components.ensemble.StackedEnsembleRegre... method*), 641
predict() (*evalml.pipelines.components.Estimator method*), 1390
predict() (*evalml.pipelines.components.estimators.ARIMARegressor method*), 861
predict() (*evalml.pipelines.components.estimators.BaselineClassifier method*), 865
predict() (*evalml.pipelines.components.estimators.BaselineRegressor method*), 868
predict() (*evalml.pipelines.components.estimators.CatBoostClassifier method*), 871
predict() (*evalml.pipelines.components.estimators.CatBoostRegressor method*), 875
predict() (*evalml.pipelines.components.estimators.classifiers.baseline_cl... method*), 646
predict() (*evalml.pipelines.components.estimators.classifiers.BaselineCla... method*), 699
predict() (*evalml.pipelines.components.estimators.classifiers.catboost_cl... method*), 649
predict() (*evalml.pipelines.components.estimators.classifiers.CatBoostCl... method*), 702
predict() (*evalml.pipelines.components.estimators.classifiers.decision_tr... method*), 653
predict() (*evalml.pipelines.components.estimators.classifiers.DecisionTre... method*), 705
predict() (*evalml.pipelines.components.estimators.classifiers.elasticnet_c... method*), 658
predict() (*evalml.pipelines.components.estimators.classifiers.ElasticNetC... method*), 709
predict() (*evalml.pipelines.components.estimators.classifiers.et_classifier... method*), 662
predict() (*evalml.pipelines.components.estimators.classifiers.ExtraTrees... method*), 713
predict() (*evalml.pipelines.components.estimators.classifiers.kneighbors_... method*), 666
predict() (*evalml.pipelines.components.estimators.classifiers.KNeighbors... method*), 716
predict() (*evalml.pipelines.components.estimators.classifiers.lightgbm_c... method*), 670
predict() (*evalml.pipelines.components.estimators.classifiers.LightGBMC... method*), 720
predict() (*evalml.pipelines.components.estimators.classifiers.logistic_reg... method*), 674
predict() (*evalml.pipelines.components.estimators.classifiers.LogisticReg... method*), 723
predict() (*evalml.pipelines.components.estimators.classifiers.RandomFor... method*), 727
predict() (*evalml.pipelines.components.estimators.classifiers.rf_classifier... method*), 677
predict() (*evalml.pipelines.components.estimators.classifiers.svm_classif...*

*method*), 681
predict() (*evalml.pipelines.components.estimators.classifiers.SVMClassifier method*), 730
predict() (*evalml.pipelines.components.estimators.classifiers.VowpalWabbitBaseClassifier method*), 685
predict() (*evalml.pipelines.components.estimators.classifiers.VowpalWabbitBinaryClassifier method*), 688
predict() (*evalml.pipelines.components.estimators.classifiers.VowpalWabbitMulticlassClassifier method*), 691
predict() (*evalml.pipelines.components.estimators.classifiers.WeightedBlendingPredictor method*), 733
predict() (*evalml.pipelines.components.estimators.classifiers.WeightedMajorityClassifier method*), 737
predict() (*evalml.pipelines.components.estimators.classifiers.XGBoostRFClassifier method*), 695
predict() (*evalml.pipelines.components.estimators.classifiers.XGBoostClassifier method*), 740
predict() (*evalml.pipelines.components.estimators.DecisionTreeClassifier method*), 878
predict() (*evalml.pipelines.components.estimators.DecisionTreeRegressor method*), 882
predict() (*evalml.pipelines.components.estimators.ElasticNetClassifier method*), 886
predict() (*evalml.pipelines.components.estimators.ElasticNetRegressor method*), 889
predict() (*evalml.pipelines.components.estimators.Estimator method*), 892
predict() (*evalml.pipelines.components.estimators.estimator_base.Estimator method*), 856
predict() (*evalml.pipelines.components.estimators.ExponentialSmoothingRegressor method*), 896
predict() (*evalml.pipelines.components.estimators.ExtraTreesClassifier method*), 899
predict() (*evalml.pipelines.components.estimators.ExtraTreesRegressor method*), 903
predict() (*evalml.pipelines.components.estimators.KNeighborsClassifier method*), 907
predict() (*evalml.pipelines.components.estimators.LightGBMClassifier method*), 911
predict() (*evalml.pipelines.components.estimators.LightGBMRegressor method*), 914
predict() (*evalml.pipelines.components.estimators.LinearRegressor method*), 917
predict() (*evalml.pipelines.components.estimators.LogisticRegressionClassifier method*), 921
predict() (*evalml.pipelines.components.estimators.ProphetRegressor method*), 924
predict() (*evalml.pipelines.components.estimators.RandomForestClassifier method*), 927
predict() (*evalml.pipelines.components.estimators.RandomForestRegressor method*), 931
predict() (*evalml.pipelines.components.estimators.regressors.baseline_regressor.BaselineRegressor method*), 745
predict() (*evalml.pipelines.components.estimators.regressors.arima.ARIMARegressor method*), 804
predict() (*evalml.pipelines.components.estimators.regressors.baseline_regressor.BaselineRegressor method*), 749
predict() (*evalml.pipelines.components.estimators.regressors.VowpalWabbitBaseRegressor method*), 808
predict() (*evalml.pipelines.components.estimators.regressors.catboost_regressor.CatBoostRegressor method*), 753
predict() (*evalml.pipelines.components.estimators.regressors.VowpalWabbitMulticlassRegressor method*), 811
predict() (*evalml.pipelines.components.estimators.regressors.decision_tree_regressor.DecisionTreeRegressor method*), 757
predict() (*evalml.pipelines.components.estimators.regressors.DecisionTreeRegressor method*), 815
predict() (*evalml.pipelines.components.estimators.regressors.elasticnet_regressor.ElasticNetRegressor method*), 760
predict() (*evalml.pipelines.components.estimators.regressors.ElasticNetRegressor method*), 818
predict() (*evalml.pipelines.components.estimators.regressors.et_regressor.ExtraTreesRegressor method*), 765
predict() (*evalml.pipelines.components.estimators.regressors.exponential_smoothing_regressor.ExponentialSmoothingRegressor method*), 769
predict() (*evalml.pipelines.components.estimators.regressors.ExponentialSmoothingRegressor method*), 822
predict() (*evalml.pipelines.components.estimators.regressors.ExtraTreesRegressor method*), 826
predict() (*evalml.pipelines.components.estimators.regressors.lightgbm_regressor.LightGBMRegressor method*), 773
predict() (*evalml.pipelines.components.estimators.regressors.LightGBMRegressor method*), 829
predict() (*evalml.pipelines.components.estimators.regressors.linear_regressor.LinearRegressor method*), 776
predict() (*evalml.pipelines.components.estimators.regressors.LinearRegressor method*), 832
predict() (*evalml.pipelines.components.estimators.regressors.prophet_regressor.ProphetRegressor method*), 781
predict() (*evalml.pipelines.components.estimators.regressors.ProphetRegressor method*), 836
predict() (*evalml.pipelines.components.estimators.regressors.RandomForestRegressor method*), 840
predict() (*evalml.pipelines.components.estimators.regressors.rf_regressor.RandomForestRegressor method*), 785
predict() (*evalml.pipelines.components.estimators.regressors.svm_regressor.SVMRegressor method*), 789
predict() (*evalml.pipelines.components.estimators.regressors.SVMRegressor method*), 843
predict() (*evalml.pipelines.components.estimators.regressors.time_series_regressor.TimeSeriesRegressor method*), 792
predict() (*evalml.pipelines.components.estimators.regressors.TimeSeriesRegressor method*), 846
predict() (*evalml.pipelines.components.estimators.regressors.vowpal_wabbit_regressor.VowpalWabbitBaseRegressor method*), 796
predict() (*evalml.pipelines.components.estimators.regressors.VowpalWabbitBaseRegressor method*), 849
predict() (*evalml.pipelines.components.estimators.regressors.xgboost_regressor.XGBoostRegressor*

method), 800
predict() (evalml.pipelines.components.estimators.regressor method), 852
predict() (evalml.pipelines.components.estimators.SVMClassifier method), 934
predict() (evalml.pipelines.components.estimators.SVMRegressor method), 937
predict() (evalml.pipelines.components.estimators.TimeSeriesBaseline method), 940
predict() (evalml.pipelines.components.estimators.VowpalWabbitB method), 943
predict() (evalml.pipelines.components.estimators.VowpalWabbitM method), 947
predict() (evalml.pipelines.components.estimators.VowpalWabbitR method), 950
predict() (evalml.pipelines.components.estimators.XGBoostClassifier method), 953
predict() (evalml.pipelines.components.estimators.XGBoostRegressor method), 956
predict() (evalml.pipelines.components.ExponentialSmoothingRegressor method), 1394
predict() (evalml.pipelines.components.ExtraTreesClassifier method), 1397
predict() (evalml.pipelines.components.ExtraTreesRegressor method), 1401
predict() (evalml.pipelines.components.KNeighborsClassifier method), 1410
predict() (evalml.pipelines.components.LightGBMClassifier method), 1416
predict() (evalml.pipelines.components.LightGBMRegressor method), 1419
predict() (evalml.pipelines.components.LinearRegressor method), 1425
predict() (evalml.pipelines.components.LogisticRegressionClassifier method), 1428
predict() (evalml.pipelines.components.ProphetRegressor method), 1459
predict() (evalml.pipelines.components.RandomForestClassifier method), 1462
predict() (evalml.pipelines.components.RandomForestRegressor method), 1465
predict() (evalml.pipelines.components.StackedEnsembleBase method), 1489
predict() (evalml.pipelines.components.StackedEnsembleClassifier method), 1493
predict() (evalml.pipelines.components.StackedEnsembleRegressor method), 1497
predict() (evalml.pipelines.components.SVMClassifier method), 1508
predict() (evalml.pipelines.components.SVMRegressor method), 1511
predict() (evalml.pipelines.components.TimeSeriesBaselineEstimator method), 1519
predict() (evalml.pipelines.components.utils.WrappedSKClassifier method), 1331
predict() (evalml.pipelines.components.utils.WrappedSKRegressor method), 1333
predict() (evalml.pipelines.components.VowpalWabbitBinaryClassifier method), 1540
predict() (evalml.pipelines.components.VowpalWabbitMulticlassClassifier method), 1543
predict() (evalml.pipelines.components.VowpalWabbitRegressor method), 1546
predict() (evalml.pipelines.components.XGBoostClassifier method), 1549
predict() (evalml.pipelines.components.XGBoostRegressor method), 1552
predict() (evalml.pipelines.DecisionTreeClassifier method), 1674
predict() (evalml.pipelines.DecisionTreeRegressor method), 1677
predict() (evalml.pipelines.ElasticNetClassifier method), 1686
predict() (evalml.pipelines.ElasticNetRegressor method), 1689
predict() (evalml.pipelines.Estimator method), 1692
predict() (evalml.pipelines.ExponentialSmoothingRegressor method), 1696
predict() (evalml.pipelines.ExtraTreesClassifier method), 1700
predict() (evalml.pipelines.ExtraTreesRegressor method), 1703
predict() (evalml.pipelines.KNeighborsClassifier method), 1712
predict() (evalml.pipelines.LightGBMClassifier method), 1716
predict() (evalml.pipelines.LightGBMRegressor method), 1719
predict() (evalml.pipelines.LinearRegressor method), 1722
predict() (evalml.pipelines.LogisticRegressionClassifier method), 1726
predict() (evalml.pipelines.multiclass_classification_pipeline.MulticlassC method), 1579
predict() (evalml.pipelines.MulticlassClassificationPipeline method), 1731
predict() (evalml.pipelines.pipeline_base.PipelineBase method), 1586
predict() (evalml.pipelines.PipelineBase method), 1746
predict() (evalml.pipelines.ProphetRegressor method), 1750
predict() (evalml.pipelines.RandomForestClassifier method), 1753
predict() (evalml.pipelines.RandomForestRegressor method), 1756
predict() (evalml.pipelines.regression_pipeline.RegressionPipeline method), 1594

predict() (evalml.pipelines.RegressionPipeline method), 1762
predict() (evalml.pipelines.StackedEnsembleBase method), 1774
predict() (evalml.pipelines.StackedEnsembleClassifier method), 1778
predict() (evalml.pipelines.StackedEnsembleRegressor method), 1782
predict() (evalml.pipelines.SVMClassifier method), 1787
predict() (evalml.pipelines.SVMRegressor method), 1791
predict() (evalml.pipelines.time_series_classification_pipelines.TimeSeriesBinaryClassificationPipeline method), 1602
predict() (evalml.pipelines.time_series_classification_pipelines.TimeSeriesClassificationPipeline method), 1609
predict() (evalml.pipelines.time_series_classification_pipelines.TimeSeriesMulticlassClassificationPipeline method), 1617
predict() (evalml.pipelines.time_series_pipeline_base.TimeSeriesPipelineBase method), 1624
predict() (evalml.pipelines.time_series_regression_pipeline.TimeSeriesRegressionPipeline method), 1633
predict() (evalml.pipelines.TimeSeriesBinaryClassificationPipeline method), 1800
predict() (evalml.pipelines.TimeSeriesClassificationPipeline method), 1807
predict() (evalml.pipelines.TimeSeriesMulticlassClassificationPipeline method), 1820
predict() (evalml.pipelines.TimeSeriesRegressionPipeline method), 1829
predict() (evalml.pipelines.VowpalWabbitBinaryClassifier method), 1839
predict() (evalml.pipelines.VowpalWabbitMulticlassClassifier method), 1842
predict() (evalml.pipelines.VowpalWabbitRegressor method), 1845
predict() (evalml.pipelines.XGBoostClassifier method), 1849
predict() (evalml.pipelines.XGBoostRegressor method), 1851
predict_in_sample() (evalml.pipelines.time_series_classification_pipelines.TimeSeriesBinaryClassificationPipeline method), 1602
predict_in_sample() (evalml.pipelines.time_series_classification_pipelines.TimeSeriesClassificationPipeline method), 1609
predict_in_sample() (evalml.pipelines.time_series_classification_pipelines.TimeSeriesMulticlassClassificationPipeline method), 1617
predict_in_sample() (evalml.pipelines.time_series_pipeline_base.TimeSeriesPipelineBase method), 1625
predict_in_sample() (evalml.pipelines.time_series_regression_pipeline.TimeSeriesRegressionPipeline method), 1634
predict_in_sample() (evalml.pipelines.TimeSeriesBinaryClassificationPipeline method), 1800
predict_in_sample() (evalml.pipelines.TimeSeriesClassificationPipeline method), 1807
predict_in_sample() (evalml.pipelines.TimeSeriesMulticlassClassificationPipeline method), 1820
predict_in_sample() (evalml.pipelines.TimeSeriesRegressionPipeline method), 1829
predict_proba() (evalml.pipelines.ARIMARegressor method), 1645
predict_proba() (evalml.pipelines.binary_classification_pipeline.BinaryClassificationPipeline method), 1650
predict_proba() (evalml.pipelines.BinaryClassificationPipeline method), 1651
predict_proba() (evalml.pipelines.CatBoostClassifier method), 1655
predict_proba() (evalml.pipelines.CatBoostRegressor method), 1658
predict_proba() (evalml.pipelines.classification_pipeline.ClassificationPipeline method), 1567
predict_proba() (evalml.pipelines.ClassificationPipeline method), 1664
predict_proba() (evalml.pipelines.components.ARIMARegressor method), 1340
predict_proba() (evalml.pipelines.components.BaselineClassifier method), 1343
predict_proba() (evalml.pipelines.components.BaselineRegressor method), 1346
predict_proba() (evalml.pipelines.components.CatBoostClassifier method), 1349
predict_proba() (evalml.pipelines.components.CatBoostRegressor method), 1353
predict_proba() (evalml.pipelines.components.DecisionTreeClassifier method), 1362
predict_proba() (evalml.pipelines.components.DecisionTreeRegressor method), 1366
predict_proba() (evalml.pipelines.components.ElasticNetClassifier method), 1382
predict_proba() (evalml.pipelines.components.ElasticNetRegressor method), 1385
predict_proba() (evalml.pipelines.components.ensemble.stacked_ensemble method), 622
predict_proba() (evalml.pipelines.components.ensemble.stacked_ensemble method), 626
predict_proba() (evalml.pipelines.components.ensemble.stacked_ensemble method), 630
predict_proba() (evalml.pipelines.components.ensemble.StackedEnsemble method), 634
predict_proba() (evalml.pipelines.components.ensemble.StackedEnsemble

*method*), 637
predict_proba() (*evalml.pipelines.components.ensemble.StackedEnsemble*
*method*), 641
predict_proba() (*evalml.pipelines.components.Estimator*
*method*), 1390
predict_proba() (*evalml.pipelines.components.estimators.ARIMA*
*method*), 862
predict_proba() (*evalml.pipelines.components.estimators.BaselineClassifier*
*method*), 865
predict_proba() (*evalml.pipelines.components.estimators.BaselineRegressor*
*method*), 868
predict_proba() (*evalml.pipelines.components.estimators.CatBoostClassifier*
*method*), 871
predict_proba() (*evalml.pipelines.components.estimators.CatBoostRegressor*
*method*), 875
predict_proba() (*evalml.pipelines.components.estimators.classifiers.BaselineClassifier*
*method*), 646
predict_proba() (*evalml.pipelines.components.estimators.classifiers.CatBoostClassifier*
*method*), 699
predict_proba() (*evalml.pipelines.components.estimators.classifiers.ElasticNetClassifier*
*method*), 649
predict_proba() (*evalml.pipelines.components.estimators.classifiers.ExtraTreesClassifier*
*method*), 702
predict_proba() (*evalml.pipelines.components.estimators.classifiers.KNeighborsClassifier*
*method*), 654
predict_proba() (*evalml.pipelines.components.estimators.classifiers.LightGBMClassifier*
*method*), 706
predict_proba() (*evalml.pipelines.components.estimators.classifiers.LinearDiscriminantAnalysis*
*method*), 658
predict_proba() (*evalml.pipelines.components.estimators.classifiers.LogisticRegressionClassifier*
*method*), 709
predict_proba() (*evalml.pipelines.components.estimators.classifiers.RandomForestClassifier*
*method*), 662
predict_proba() (*evalml.pipelines.components.estimators.classifiers.StackedEnsembleClassifier*
*method*), 713
predict_proba() (*evalml.pipelines.components.estimators.classifiers.VowpalWabbitBinaryClassifier*
*method*), 666
predict_proba() (*evalml.pipelines.components.estimators.classifiers.VowpalWabbitMulticlassClassifier*
*method*), 716
predict_proba() (*evalml.pipelines.components.estimators.classifiers.XGBoostClassifier*
*method*), 670
predict_proba() (*evalml.pipelines.components.estimators.classifiers.XGBoostRFClassifier*
*method*), 720
predict_proba() (*evalml.pipelines.components.estimators.regressors.ElasticNetRegressor*
*method*), 674
predict_proba() (*evalml.pipelines.components.estimators.regressors.ExtraTreesRegressor*
*method*), 723
predict_proba() (*evalml.pipelines.components.estimators.regressors.KNeighborsRegressor*
*method*), 727
predict_proba() (*evalml.pipelines.components.estimators.regressors.LightGBMRegressor*
*method*), 678
predict_proba() (*evalml.pipelines.components.estimators.regressors.LinearRegressor*
*method*), 681
predict_proba() (*evalml.pipelines.components.estimators.regressors.RandomForestRegressor*
*method*), 730
predict_proba() (*evalml.pipelines.components.estimators.classifiers.VowpalWabbitBinaryClassifier*
*method*), 685
predict_proba() (*evalml.pipelines.components.estimators.classifiers.VowpalWabbitMulticlassClassifier*
*method*), 688
predict_proba() (*evalml.pipelines.components.estimators.classifiers.VowpalWabbitBaseClassifier*
*method*), 691
predict_proba() (*evalml.pipelines.components.estimators.classifiers.VowpalWabbitBinaryClassifier*
*method*), 733
predict_proba() (*evalml.pipelines.components.estimators.classifiers.VowpalWabbitMulticlassClassifier*
*method*), 737
predict_proba() (*evalml.pipelines.components.estimators.classifiers.xgboost.XGBoostClassifier*
*method*), 695
predict_proba() (*evalml.pipelines.components.estimators.classifiers.XGBoostRFClassifier*
*method*), 740
predict_proba() (*evalml.pipelines.components.estimators.DecisionTreeClassifier*
*method*), 879
predict_proba() (*evalml.pipelines.components.estimators.DecisionTreeRegressor*
*method*), 883
predict_proba() (*evalml.pipelines.components.estimators.ElasticNetClassifier*
*method*), 886
predict_proba() (*evalml.pipelines.components.estimators.ElasticNetRegressor*
*method*), 889
predict_proba() (*evalml.pipelines.components.estimators.ExtraTreesClassifier*
*method*), 892
predict_proba() (*evalml.pipelines.components.estimator.Estimator*
*method*), 857
predict_proba() (*evalml.pipelines.components.estimators.ExponentialSmoothing*
*method*), 896
predict_proba() (*evalml.pipelines.components.estimators.ExtraTreesClassifier*
*method*), 900
predict_proba() (*evalml.pipelines.components.estimators.ExtraTreesRegressor*
*method*), 903
predict_proba() (*evalml.pipelines.components.estimators.KNeighborsClassifier*
*method*), 907
predict_proba() (*evalml.pipelines.components.estimators.KNeighborsRegressor*
*method*), 911
predict_proba() (*evalml.pipelines.components.estimators.LightGBMClassifier*
*method*), 914
predict_proba() (*evalml.pipelines.components.estimators.LightGBMRegressor*
*method*), 917
predict_proba() (*evalml.pipelines.components.estimators.LinearRegressor*
*method*), 921
predict_proba() (*evalml.pipelines.components.estimators.LogisticRegressionClassifier*
*method*), 924
predict_proba() (*evalml.pipelines.components.estimators.LogisticRegressionCV*
*method*), 928
predict_proba() (*evalml.pipelines.components.estimators.RandomForestClassifier*
*method*), 931
predict_proba() (*evalml.pipelines.components.estimators.RandomForestRegressor*
*method*), 745
predict_proba() (*evalml.pipelines.components.estimators.regressors.ARIMA*
*method*), 805
predict_proba() (*evalml.pipelines.components.estimators.regressors.baseline.BaselineRegressor*

*method*), 749
predict_proba() (*evalml.pipelines.components.estimators.*
*method*), 808
predict_proba() (*evalml.pipelines.components.estimators.*
*method*), 753
predict_proba() (*evalml.pipelines.components.estimators.*
*method*), 811
predict_proba() (*evalml.pipelines.components.estimators.*
*method*), 757
predict_proba() (*evalml.pipelines.components.estimators.*
*method*), 815
predict_proba() (*evalml.pipelines.components.estimators.*
*method*), 760
predict_proba() (*evalml.pipelines.components.estimators.*
*method*), 818
predict_proba() (*evalml.pipelines.components.estimators.*
*method*), 765
predict_proba() (*evalml.pipelines.components.estimators.*
*method*), 769
predict_proba() (*evalml.pipelines.components.estimators.*
*method*), 822
predict_proba() (*evalml.pipelines.components.estimators.*
*method*), 826
predict_proba() (*evalml.pipelines.components.estimators.*
*method*), 773
predict_proba() (*evalml.pipelines.components.estimators.*
*method*), 829
predict_proba() (*evalml.pipelines.components.estimators.*
*method*), 776
predict_proba() (*evalml.pipelines.components.estimators.*
*method*), 832
predict_proba() (*evalml.pipelines.components.estimators.*
*method*), 781
predict_proba() (*evalml.pipelines.components.estimators.*
*method*), 836
predict_proba() (*evalml.pipelines.components.estimators.*
*method*), 840
predict_proba() (*evalml.pipelines.components.estimators.*
*method*), 785
predict_proba() (*evalml.pipelines.components.estimators.*
*method*), 789
predict_proba() (*evalml.pipelines.components.estimators.*
*method*), 843
predict_proba() (*evalml.pipelines.components.estimators.*
*method*), 793
predict_proba() (*evalml.pipelines.components.estimators.*
*method*), 846
predict_proba() (*evalml.pipelines.components.estimators.*
*method*), 796
predict_proba() (*evalml.pipelines.components.estimators.*
*method*), 849
predict_proba() (*evalml.pipelines.components.estimators.*
*method*), 800
predict_proba() (*evalml.pipelines.components.estimators.*
*method*), 853
predict_proba() (*evalml.pipelines.components.estimators.SVMClassifier*
*method*), 934
predict_proba() (*evalml.pipelines.components.estimators.SVMRegressor*
*method*), 937
predict_proba() (*evalml.pipelines.components.estimators.TimeSeriesBas*
*method*), 940
predict_proba() (*evalml.pipelines.components.estimators.VowpalWabbit*
*method*), 943
predict_proba() (*evalml.pipelines.components.estimators.VowpalWabbit*
*method*), 947
predict_proba() (*evalml.pipelines.components.estimators.VowpalWabbit*
*method*), 950
predict_proba() (*evalml.pipelines.components.estimators.XGBoostClass*
*method*), 953
predict_proba() (*evalml.pipelines.components.estimators.XGBoostRegre*
*method*), 956
predict_proba() (*evalml.pipelines.regression_pipelines.ExponentialSmoothingRe*
*method*), 1394
predict_proba() (*evalml.pipelines.components.ExtraTreesClassifier*
*method*), 1398
predict_proba() (*evalml.pipelines.components.ExtraTreesRegressor*
*method*), 1401
predict_proba() (*evalml.pipelines.components.KNeighborsClassifier*
*method*), 1410
predict_proba() (*evalml.pipelines.components.LightGBMClassifier*
*method*), 1416
predict_proba() (*evalml.pipelines.components.LightGBMRegressor*
*method*), 1419
predict_proba() (*evalml.pipelines.components.LinearRegressor*
*method*), 1425
predict_proba() (*evalml.pipelines.components.LogisticRegressionClassi*
*method*), 1428
predict_proba() (*evalml.pipelines.components.ProphetRegressor*
*method*), 1459
predict_proba() (*evalml.pipelines.components.RandomForestClassifier*
*method*), 1462
predict_proba() (*evalml.pipelines.components.RandomForestRegressor*
*method*), 1465
predict_proba() (*evalml.pipelines.components.StackedEnsembleBase*
*method*), 1490
predict_proba() (*evalml.pipelines.components.StackedEnsembleClassifi*
*method*), 1493
predict_proba() (*evalml.pipelines.components.StackedEnsembleRegress*
*method*), 1497
predict_proba() (*evalml.pipelines.components.SVMClassifier*
*method*), 1508
predict_proba() (*evalml.pipelines.components.SVMRegressor*
*method*), 1511
predict_proba() (*evalml.pipelines.components.TimeSeriesBaselineEstim*
*method*), 1520
predict_proba() (*evalml.pipelines.utils.WrappedSKClassifie*
*method*), 1332
predict_proba() (*evalml.pipelines.components.VowpalWabbitBinaryCla*

*method*), 1540
predict_proba() (*evalml.pipelines.components.VowpalWabbitMulticlassClassifier method*), 1543
predict_proba() (*evalml.pipelines.components.VowpalWabbitRegressor method*), 1546
predict_proba() (*evalml.pipelines.components.XGBoostClassifier method*), 1549
predict_proba() (*evalml.pipelines.components.XGBoostRegressor method*), 1552
predict_proba() (*evalml.pipelines.DecisionTreeClassifier method*), 1674
predict_proba() (*evalml.pipelines.DecisionTreeRegressor method*), 1678
predict_proba() (*evalml.pipelines.ElasticNetClassifier method*), 1686
predict_proba() (*evalml.pipelines.ElasticNetRegressor method*), 1689
predict_proba() (*evalml.pipelines.Estimator method*), 1693
predict_proba() (*evalml.pipelines.ExponentialSmoothingRegressor method*), 1696
predict_proba() (*evalml.pipelines.ExtraTreesClassifier method*), 1700
predict_proba() (*evalml.pipelines.ExtraTreesRegressor method*), 1703
predict_proba() (*evalml.pipelines.KNeighborsClassifier method*), 1712
predict_proba() (*evalml.pipelines.LightGBMClassifier method*), 1716
predict_proba() (*evalml.pipelines.LightGBMRegressor method*), 1719
predict_proba() (*evalml.pipelines.LinearRegressor method*), 1722
predict_proba() (*evalml.pipelines.LogisticRegressionClassifier method*), 1726
predict_proba() (*evalml.pipelines.multiclass_classification_pipeline.MulticlassClassificationPipeline method*), 1580
predict_proba() (*evalml.pipelines.MulticlassClassificationPipeline method*), 1732
predict_proba() (*evalml.pipelines.ProphetRegressor method*), 1750
predict_proba() (*evalml.pipelines.RandomForestClassifier method*), 1754
predict_proba() (*evalml.pipelines.RandomForestRegressor method*), 1756
predict_proba() (*evalml.pipelines.StackedEnsembleBase method*), 1775
predict_proba() (*evalml.pipelines.StackedEnsembleClassifier method*), 1778
predict_proba() (*evalml.pipelines.StackedEnsembleRegressor method*), 1782
predict_proba() (*evalml.pipelines.SVMClassifier method*), 1787
predict_proba() (*evalml.pipelines.SVMRegressor method*), 1791
predict_proba() (*evalml.pipelines.time_series_classification_pipelines.T method*), 1602
predict_proba() (*evalml.pipelines.time_series_classification_pipelines.T method*), 1610
predict_proba() (*evalml.pipelines.time_series_classification_pipelines.T method*), 1617
predict_proba() (*evalml.pipelines.TimeSeriesBinaryClassificationPipeli method*), 1800
predict_proba() (*evalml.pipelines.TimeSeriesClassificationPipeline method*), 1807
predict_proba() (*evalml.pipelines.TimeSeriesMulticlassClassificationPi method*), 1821
predict_proba() (*evalml.pipelines.VowpalWabbitBinaryClassifier method*), 1839
predict_proba() (*evalml.pipelines.VowpalWabbitMulticlassClassifier method*), 1842
predict_proba() (*evalml.pipelines.VowpalWabbitRegressor method*), 1845
predict_proba() (*evalml.pipelines.XGBoostClassifier method*), 1849
predict_proba() (*evalml.pipelines.XGBoostRegressor method*), 1852
predict_proba_in_sample() (*evalml.pipelines.time_series_classification_pipelines.TimeSeries method*), 1603
predict_proba_in_sample() (*evalml.pipelines.time_series_classification_pipelines.TimeSeries method*), 1610
predict_proba_in_sample() (*evalml.pipelines.time_series_classification_pipelines.TimeSeries method*), 1618
predict_proba_in_sample() (*evalml.pipelines.TimeSeriesBinaryClassificationPipeline method*), 1801
predict_proba_in_sample() (*evalml.pipelines.TimeSeriesClassificationPipeline method*), 1808
predict_proba_in_sample() (*evalml.pipelines.TimeSeriesMulticlassClassificationPipeline method*), 1821
print_deps() (*in module evalml.utils.cli_utils*), 1900
print_info() (*in module evalml.utils.cli_utils*), 1900
print_sys_info() (*in module evalml.utils.cli_utils*), 1900
ProblemTypes (*class in evalml.problem_types*), 1882
ProblemTypes (*class in evalml.problem_types.problem_types*), 1876
Progress (*class in evalml.automl*), 281
Progress (*class in evalml.automl.progress*), 268
ProphetRegressor (*class in evalml.pipelines*), 1747
ProphetRegressor (*class in evalml.pipelines.components*), 1456
ProphetRegressor (*class in*

*evalml.pipelines.components.estimators*), 921
ProphetRegressor (*class in evalml.pipelines.components.estimators.regressors*), 833
ProphetRegressor (*class in evalml.pipelines.components.estimators.regressors.prophet_regressor*), 777
propose() (*evalml.tuners.grid_search_tuner.GridSearchTuner method*), 1885
propose() (*evalml.tuners.GridSearchTuner method*), 1893
propose() (*evalml.tuners.random_search_tuner.RandomSearchTuner method*), 1887
propose() (*evalml.tuners.RandomSearchTuner method*), 1894
propose() (*evalml.tuners.skopt_tuner.SKOptTuner method*), 1889
propose() (*evalml.tuners.SKOptTuner method*), 1896
propose() (*evalml.tuners.Tuner method*), 1897
propose() (*evalml.tuners.tuner.Tuner method*), 1891

## R

R2 (*class in evalml.objectives*), 602
R2 (*class in evalml.objectives.standard_metrics*), 515
raise_error_callback() (*in module evalml.automl.callbacks*), 266
RandomForestClassifier (*class in evalml.pipelines*), 1751
RandomForestClassifier (*class in evalml.pipelines.components*), 1459
RandomForestClassifier (*class in evalml.pipelines.components.estimators*), 925
RandomForestClassifier (*class in evalml.pipelines.components.estimators.classifiers*), 724
RandomForestClassifier (*class in evalml.pipelines.components.estimators.classifiers.rf_classifier*), 675
RandomForestRegressor (*class in evalml.pipelines*), 1754
RandomForestRegressor (*class in evalml.pipelines.components*), 1462
RandomForestRegressor (*class in evalml.pipelines.components.estimators*), 928
RandomForestRegressor (*class in evalml.pipelines.components.estimators.regressors*), 837
RandomForestRegressor (*class in evalml.pipelines.components.estimators.regressors.rf_regressor*), 782
RandomSearchTuner (*class in evalml.tuners*), 1893
RandomSearchTuner (*class in evalml.tuners.random_search_tuner*), 1885
ranking_only_objectives() (*in module evalml.objectives*), 603
ranking_only_objectives() (*in module evalml.objectives.utils*), 532
rankings (*evalml.automl.automl_search.AutoMLSearch property*), 262
rankings (*evalml.automl.AutoMLSearch property*), 278
rankings (*evalml.AutoMLSearch property*), 1918
readable_explanation() (*in module evalml.model_understanding.feature_explanations*), 411
Recall (*class in evalml.objectives*), 603
Recall (*class in evalml.objectives.standard_metrics*), 517
RecallMacro (*class in evalml.objectives*), 606
RecallMacro (*class in evalml.objectives.standard_metrics*), 519
RecallMicro (*class in evalml.objectives*), 607
RecallMicro (*class in evalml.objectives.standard_metrics*), 521
RecallWeighted (*class in evalml.objectives*), 609
RecallWeighted (*class in evalml.objectives.standard_metrics*), 522
RecursiveFeatureEliminationSelector (*class in evalml.pipelines.components.transformers.feature_selection.recur*, 1000
register() (*evalml.pipelines.components.component_base_meta.Compon method*), 1327
register() (*evalml.pipelines.components.ComponentBaseMeta method*), 1356
register() (*evalml.pipelines.components.transformers.encoders.onehot_e method*), 975
register() (*evalml.pipelines.components.transformers.encoders.ordinal_ method*), 980
register() (*evalml.pipelines.components.transformers.imputers.target_im method*), 1048
register() (*evalml.pipelines.pipeline_meta.PipelineBaseMeta method*), 1588
register() (*evalml.utils.base_meta.BaseMeta method*), 1898
RegressionObjective (*class in evalml.objectives*), 611
RegressionObjective (*class in evalml.objectives.regression_objective*), 457
RegressionPipeline (*class in evalml.pipelines*), 1757
RegressionPipeline (*class in evalml.pipelines.regression_pipeline*), 1589
ReplaceNullableTypes (*class in evalml.pipelines.components*), 1465
ReplaceNullableTypes (*class in evalml.pipelines.components.transformers*), 1271
ReplaceNullableTypes (*class in*

*evalml.pipelines.components.transformers.preprocessing*), 1160
ReplaceNullableTypes (*class in evalml.pipelines.components.transformers.preprocessing.replace_nullable_types*), 1103
resplit_training_data() (*in module evalml.automl*), 282
resplit_training_data() (*in module evalml.automl.utils*), 272
results (*evalml.automl.automl_search.AutoMLSearch property*), 262
results (*evalml.automl.AutoMLSearch property*), 278
results (*evalml.AutoMLSearch property*), 1918
return_progress() (*evalml.automl.Progress method*), 281
return_progress() (*evalml.automl.progress.Progress method*), 269
RFClassifierRFESelector (*class in evalml.pipelines.components*), 1468
RFClassifierRFESelector (*class in evalml.pipelines.components.transformers*), 1273
RFClassifierRFESelector (*class in evalml.pipelines.components.transformers.feature_selection*), 1019
RFClassifierRFESelector (*class in evalml.pipelines.components.transformers.feature_selection.recursive_feature_elimination_selector*), 1003
RFClassifierSelectFromModel (*class in evalml.pipelines*), 1763
RFClassifierSelectFromModel (*class in evalml.pipelines.components*), 1471
RFClassifierSelectFromModel (*class in evalml.pipelines.components.transformers*), 1276
RFClassifierSelectFromModel (*class in evalml.pipelines.components.transformers.feature_selection*), 1023
RFClassifierSelectFromModel (*class in evalml.pipelines.components.transformers.feature_selection.rf_classifier_feature_selector*), 1009
RFRegressorRFESelector (*class in evalml.pipelines.components*), 1474
RFRegressorRFESelector (*class in evalml.pipelines.components.transformers*), 1279
RFRegressorRFESelector (*class in evalml.pipelines.components.transformers.feature_selection*), 1026
RFRegressorRFESelector (*class in evalml.pipelines.components.transformers.feature_selection.recursive_feature_elimination_selector*), 1006
RFRegressorSelectFromModel (*class in evalml.pipelines*), 1766
RFRegressorSelectFromModel (*class in evalml.pipelines.components*), 1477
RFRegressorSelectFromModel (*class in evalml.pipelines.components.transformers*), 1282
RFRegressorSelectFromModel (*class in evalml.pipelines.components.transformers.feature_selection*), 1029
RFRegressorSelectFromModel (*class in evalml.pipelines.components.transformers.feature_selection.rf_re*, 1013
roc_curve() (*in module evalml.model_understanding*), 438
roc_curve() (*in module evalml.model_understanding.metrics*), 415
RootMeanSquaredError (*class in evalml.objectives*), 613
RootMeanSquaredError (*class in evalml.objectives.standard_metrics*), 524
RootMeanSquaredLogError (*class in evalml.objectives*), 614
RootMeanSquaredLogError (*class in evalml.objectives.standard_metrics*), 526
rule_of_interest() (*in module evalml.pipelines.utils*), 1639

## S

safe_repr() (*in module evalml.utils*), 1913
safe_repr() (*in module evalml.utils.gen_utils*), 1905
save() (*evalml.automl.automl_search.AutoMLSearch method*), 263
save() (*evalml.automl.AutoMLSearch method*), 278
save() (*evalml.AutoMLSearch method*), 1919
save() (*evalml.pipelines.ARIMARegressor method*), 1645
save() (*evalml.pipelines.binary_classification_pipeline.BinaryClassificati method*), 1559
save() (*evalml.pipelines.BinaryClassificationPipeline method*), 1651
save() (*evalml.pipelines.CatBoostClassifier method*), 1655
save() (*evalml.pipelines.CatBoostRegressor method*), 1658
save() (*evalml.pipelines.classification_pipeline.ClassificationPipeline method*), 1567
save() (*evalml.pipelines.ClassificationPipeline method*), 1664
save() (*evalml.pipelines.components.ARIMARegressor method*), 1340
save() (*evalml.pipelines.components.BaselineClassifier method*), 1343
save() (*evalml.pipelines.components.BaselineRegressor method*), 1346

save() (*evalml.pipelines.components.CatBoostClassifier method*), 1350
save() (*evalml.pipelines.components.CatBoostRegressor method*), 1353
save() (*evalml.pipelines.components.component_base.ComponentBase method*), 1326
save() (*evalml.pipelines.components.ComponentBase method*), 1355
save() (*evalml.pipelines.components.DateTimeFeaturizer method*), 1358
save() (*evalml.pipelines.components.DecisionTreeClassifier method*), 1362
save() (*evalml.pipelines.components.DecisionTreeRegressor method*), 1366
save() (*evalml.pipelines.components.DFSTransformer method*), 1369
save() (*evalml.pipelines.components.DropColumns method*), 1371
save() (*evalml.pipelines.components.DropNaNRowsTransformer method*), 1373
save() (*evalml.pipelines.components.DropNullColumns method*), 1376
save() (*evalml.pipelines.components.DropRowsTransformer method*), 1378
save() (*evalml.pipelines.components.ElasticNetClassifier method*), 1382
save() (*evalml.pipelines.components.ElasticNetRegressor method*), 1385
save() (*evalml.pipelines.components.EmailFeaturizer method*), 1387
save() (*evalml.pipelines.components.ensemble.stacked_ensemble_base.StackedEnsembleBase method*), 622
save() (*evalml.pipelines.components.ensemble.stacked_ensemble_classifier.StackedEnsembleClassifier method*), 626
save() (*evalml.pipelines.components.ensemble.stacked_ensemble_regressor.StackedEnsembleRegressor method*), 630
save() (*evalml.pipelines.components.ensemble.StackedEnsembleBase method*), 634
save() (*evalml.pipelines.components.ensemble.StackedEnsembleClassifier method*), 638
save() (*evalml.pipelines.components.ensemble.StackedEnsembleRegressor method*), 642
save() (*evalml.pipelines.components.Estimator method*), 1390
save() (*evalml.pipelines.components.estimators.ARIMARegressor method*), 862
save() (*evalml.pipelines.components.estimators.BaselineClassifier method*), 865
save() (*evalml.pipelines.components.estimators.BaselineRegressor method*), 868
save() (*evalml.pipelines.components.estimators.CatBoostClassifier method*), 872
save() (*evalml.pipelines.components.estimators.CatBoostRegressor method*), 875
save() (*evalml.pipelines.components.estimators.classifiers.baseline_classi method*), 646
save() (*evalml.pipelines.components.estimators.classifiers.BaselineClassif method*), 699
save() (*evalml.pipelines.components.estimators.classifiers.catboost_classi method*), 650
save() (*evalml.pipelines.components.estimators.classifiers.CatBoostClassi method*), 702
save() (*evalml.pipelines.components.estimators.classifiers.decision_tree_c method*), 654
save() (*evalml.pipelines.components.estimators.classifiers.DecisionTreeCl method*), 706
save() (*evalml.pipelines.components.estimators.classifiers.elasticnet_class method*), 658
save() (*evalml.pipelines.components.estimators.classifiers.ElasticNetClass method*), 709
save() (*evalml.pipelines.components.estimators.classifiers.et_classifier.Ext method*), 662
save() (*evalml.pipelines.components.estimators.classifiers.ExtraTreesClas method*), 713
save() (*evalml.pipelines.components.estimators.classifiers.kneighbors_cla method*), 666
save() (*evalml.pipelines.components.estimators.classifiers.KNeighborsCla method*), 717
save() (*evalml.pipelines.components.estimators.classifiers.lightgbm_class method*), 670
save() (*evalml.pipelines.components.estimators.classifiers.LightGBMClas method*), 720
save() (*evalml.pipelines.components.estimators.classifiers.logistic_regress method*), 674
save() (*evalml.pipelines.components.estimators.classifiers.LogisticRegress method*), 724
save() (*evalml.pipelines.components.estimators.classifiers.RandomForestC method*), 727
save() (*evalml.pipelines.components.estimators.classifiers.rf_classifier.Ra method*), 678
save() (*evalml.pipelines.components.estimators.classifiers.svm_classifier.S method*), 681
save() (*evalml.pipelines.components.estimators.classifiers.SVMClassifier method*), 730
save() (*evalml.pipelines.components.estimators.classifiers.vowpal_wabbit_ method*), 685
save() (*evalml.pipelines.components.estimators.classifiers.vowpal_wabbit_ method*), 688
save() (*evalml.pipelines.components.estimators.classifiers.vowpal_wabbit_ method*), 692
save() (*evalml.pipelines.components.estimators.classifiers.VowpalWabbitB method*), 734
save() (*evalml.pipelines.components.estimators.classifiers.VowpalWabbitM method*), 737
save() (*evalml.pipelines.components.estimators.classifiers.xgboost_classif method*), 695
save() (*evalml.pipelines.components.estimators.classifiers.XGBoostClassi method*), 740

save() (*evalml.pipelines.components.estimators.DecisionTreeClassifier method*), 879
save() (*evalml.pipelines.components.estimators.DecisionTreeRegressor method*), 883
save() (*evalml.pipelines.components.estimators.ElasticNetClassifier method*), 886
save() (*evalml.pipelines.components.estimators.ElasticNetRegressor method*), 889
save() (*evalml.pipelines.components.estimators.Estimator method*), 892
save() (*evalml.pipelines.components.estimators.estimator.Estimator method*), 857
save() (*evalml.pipelines.components.ExponentialSmoothingRegressor method*), 896
save() (*evalml.pipelines.components.estimators.ExtraTreesClassifier method*), 900
save() (*evalml.pipelines.components.estimators.ExtraTreesRegressor method*), 904
save() (*evalml.pipelines.components.estimators.KNeighborsClassifier method*), 907
save() (*evalml.pipelines.components.estimators.LightGBMClassifier method*), 911
save() (*evalml.pipelines.components.estimators.LightGBMRegressor method*), 915
save() (*evalml.pipelines.components.estimators.LinearRegressor method*), 918
save() (*evalml.pipelines.components.estimators.LogisticRegressionClassifier method*), 921
save() (*evalml.pipelines.components.estimators.ProphetRegressor method*), 925
save() (*evalml.pipelines.components.estimators.RandomForestClassifier method*), 928
save() (*evalml.pipelines.components.estimators.RandomForestRegressor method*), 931
save() (*evalml.pipelines.components.regressors.arima_regressor.ARIMARegressor method*), 746
save() (*evalml.pipelines.components.estimators.regressors.arima_regressor.ARIMARegressor method*), 805
save() (*evalml.pipelines.components.estimators.regressors.baseline_regressor.BaselineRegressor method*), 749
save() (*evalml.pipelines.components.estimators.regressors.baseline_regressor.BaselineRegressor method*), 808
save() (*evalml.pipelines.components.estimators.regressors.catboost_regressor.CatBoostRegressor method*), 753
save() (*evalml.pipelines.components.estimators.regressors.catboost_regressor.CatBoostRegressor method*), 812
save() (*evalml.pipelines.components.estimators.regressors.decision_tree_regressor.DecisionTreeRegressor method*), 757
save() (*evalml.pipelines.components.estimators.regressors.elastic_net_regressor.ElasticNetRegressor method*), 815
save() (*evalml.pipelines.components.estimators.regressors.elastic_net_regressor.ElasticNetRegressor method*), 761
save() (*evalml.pipelines.components.estimators.regressors.elastic_net_regressor.ElasticNetRegressor method*), 819
save() (*evalml.pipelines.components.estimators.regressors.et_regressor.ExtraTreesRegressor method*), 765
save() (*evalml.pipelines.components.estimators.regressors.exponential_smoothing_regressor.ExponentialSmoothingRegressor method*), 769
save() (*evalml.pipelines.components.estimators.regressors.ExponentialSmoothingRegressor method*), 822
save() (*evalml.pipelines.components.estimators.regressors.ExtraTreesRegressor method*), 826
save() (*evalml.pipelines.components.estimators.regressors.lightgbm_regressor.LightGBMRegressor method*), 773
save() (*evalml.pipelines.components.estimators.regressors.LightGBMRegressor method*), 830
save() (*evalml.pipelines.components.regressors.linear_regressor.LinearRegressor method*), 777
save() (*evalml.pipelines.components.estimators.regressors.LinearRegressor method*), 833
save() (*evalml.pipelines.components.estimators.regressors.prophet_regressor.ProphetRegressor method*), 782
save() (*evalml.pipelines.components.estimators.regressors.ProphetRegressor method*), 837
save() (*evalml.pipelines.components.estimators.regressors.RandomForestRegressor method*), 840
save() (*evalml.pipelines.components.estimators.regressors.rf_regressor.RandomForestRegressor method*), 786
save() (*evalml.pipelines.components.estimators.regressors.svm_regressor.SVMRegressor method*), 789
save() (*evalml.pipelines.components.estimators.regressors.SVMRegressor method*), 843
save() (*evalml.pipelines.components.estimators.regressors.time_series_baseline_regressor.TimeSeriesBaselineRegressor method*), 793
save() (*evalml.pipelines.components.estimators.regressors.TimeSeriesBaselineRegressor method*), 846
save() (*evalml.pipelines.components.estimators.regressors.vowpal_wabbit_regressor.VowpalWabbitRegressor method*), 796
save() (*evalml.pipelines.components.estimators.regressors.VowpalWabbitRegressor method*), 850
save() (*evalml.pipelines.components.estimators.regressors.xgboost_regressor.XGBoostRegressor method*), 800
save() (*evalml.pipelines.components.estimators.regressors.XGBoostRegressor method*), 853
save() (*evalml.pipelines.components.SVMClassifier method*), 934
save() (*evalml.pipelines.components.estimators.SVMRegressor method*), 937
save() (*evalml.pipelines.components.estimators.TimeSeriesBaselineEstimator method*), 940
save() (*evalml.pipelines.components.estimators.VowpalWabbitBinaryClassifier method*), 944
save() (*evalml.pipelines.components.estimators.VowpalWabbitMulticlassClassifier method*), 947
save() (*evalml.pipelines.components.estimators.VowpalWabbitRegressor method*), 950
save() (*evalml.pipelines.components.estimators.XGBoostClassifier method*), 953

save() (*evalml.pipelines.components.estimators.XGBoostRegressor method*), 956
save() (*evalml.pipelines.components.ExponentialSmoothingRegressor method*), 1394
save() (*evalml.pipelines.components.ExtraTreesClassifier method*), 1398
save() (*evalml.pipelines.components.ExtraTreesRegressor method*), 1401
save() (*evalml.pipelines.components.FeatureSelector method*), 1404
save() (*evalml.pipelines.components.Imputer method*), 1406
save() (*evalml.pipelines.components.KNeighborsClassifier method*), 1410
save() (*evalml.pipelines.components.LabelEncoder method*), 1412
save() (*evalml.pipelines.components.LightGBMClassifier method*), 1416
save() (*evalml.pipelines.components.LightGBMRegressor method*), 1420
save() (*evalml.pipelines.components.LinearDiscriminantAnalysis method*), 1422
save() (*evalml.pipelines.components.LinearRegressor method*), 1425
save() (*evalml.pipelines.components.LogisticRegressionClassifier method*), 1429
save() (*evalml.pipelines.components.LogTransformer method*), 1431
save() (*evalml.pipelines.components.LSA method*), 1433
save() (*evalml.pipelines.components.NaturalLanguageFeaturizer method*), 1436
save() (*evalml.pipelines.components.OneHotEncoder method*), 1439
save() (*evalml.pipelines.components.OrdinalEncoder method*), 1443
save() (*evalml.pipelines.components.Oversampler method*), 1445
save() (*evalml.pipelines.components.PCA method*), 1448
save() (*evalml.pipelines.components.PerColumnImputer method*), 1450
save() (*evalml.pipelines.components.PolynomialDecomposer method*), 1455
save() (*evalml.pipelines.components.ProphetRegressor method*), 1459
save() (*evalml.pipelines.components.RandomForestClassifier method*), 1462
save() (*evalml.pipelines.components.RandomForestRegressor method*), 1465
save() (*evalml.pipelines.components.ReplaceNullableTypes method*), 1467
save() (*evalml.pipelines.components.RFClassifierRFESelector method*), 1470
save() (*evalml.pipelines.components.RFClassifierSelectFromModel method*), 1473
save() (*evalml.pipelines.components.RFRegressorRFESelector method*), 1476
save() (*evalml.pipelines.components.RFRegressorSelectFromModel method*), 1479
save() (*evalml.pipelines.components.SelectByType method*), 1481
save() (*evalml.pipelines.components.SelectColumns method*), 1484
save() (*evalml.pipelines.components.SimpleImputer method*), 1486
save() (*evalml.pipelines.components.StackedEnsembleBase method*), 1490
save() (*evalml.pipelines.components.StackedEnsembleClassifier method*), 1494
save() (*evalml.pipelines.components.StackedEnsembleRegressor method*), 1497
save() (*evalml.pipelines.components.StandardScaler method*), 1499
save() (*evalml.pipelines.components.STLDecomposer method*), 1504
save() (*evalml.pipelines.components.SVMClassifier method*), 1508
save() (*evalml.pipelines.components.SVMRegressor method*), 1511
save() (*evalml.pipelines.components.TargetEncoder method*), 1514
save() (*evalml.pipelines.components.TargetImputer method*), 1516
save() (*evalml.pipelines.components.TimeSeriesBaselineEstimator method*), 1520
save() (*evalml.pipelines.components.TimeSeriesFeaturizer method*), 1522
save() (*evalml.pipelines.components.TimeSeriesImputer method*), 1525
save() (*evalml.pipelines.components.TimeSeriesRegularizer method*), 1528
save() (*evalml.pipelines.components.Transformer method*), 1531
save() (*evalml.pipelines.components.transformers.column_selectors.Colu method*), 1203
save() (*evalml.pipelines.components.transformers.column_selectors.Drop method*), 1205
save() (*evalml.pipelines.components.transformers.column_selectors.Selec method*), 1208
save() (*evalml.pipelines.components.transformers.column_selectors.Selec method*), 1210
save() (*evalml.pipelines.components.transformers.DateTimeFeaturizer method*), 1218
save() (*evalml.pipelines.components.transformers.DFSTransformer method*), 1220
save() (*evalml.pipelines.components.transformers.dimensionality_reducti method*), 959

save() (*evalml.pipelines.components.transformers.dimensionality_reduction.LinearDiscriminantAnalysis method*), 965
save() (*evalml.pipelines.components.transformers.dimensionality_reduction.PCA method*), 967
save() (*evalml.pipelines.components.transformers.dimensionality_reduction.LSA method*), 962
save() (*evalml.pipelines.components.transformers.DropColumns method*), 1223
save() (*evalml.pipelines.components.transformers.DropNaRowsTransformer method*), 1225
save() (*evalml.pipelines.components.transformers.DropNullColumns method*), 1228
save() (*evalml.pipelines.components.transformers.DropRowsTransformer method*), 1230
save() (*evalml.pipelines.components.transformers.EmailFeaturizer method*), 1232
save() (*evalml.pipelines.components.transformers.encoders.base_encoder.BaseEncoder method*), 970
save() (*evalml.pipelines.components.transformers.encoders.LabelEncoder method*), 986
save() (*evalml.pipelines.components.transformers.encoders.one_hot_encoder.OneHotEncoder method*), 974
save() (*evalml.pipelines.components.transformers.encoders.OrdinalEncoder method*), 989
save() (*evalml.pipelines.components.transformers.encoders.ordinal_encoder.OrdinalEncoder method*), 979
save() (*evalml.pipelines.components.transformers.encoders.OneHotEncoder method*), 993
save() (*evalml.pipelines.components.transformers.encoders.target_encoder.TargetEncoder method*), 983
save() (*evalml.pipelines.components.transformers.encoders.TargetEncoder method*), 995
save() (*evalml.pipelines.components.transformers.feature_selection.base_feature_selector.BaseFeatureSelector method*), 999
save() (*evalml.pipelines.components.transformers.feature_selection.FeatureSelection method*), 1019
save() (*evalml.pipelines.components.transformers.feature_selection.recursive_feature_elimination_cv_feature_selector.RecursiveFeatureElimination method*), 1002
save() (*evalml.pipelines.components.transformers.feature_selection.recursive_feature_elimination_feature_selector.RFClassifierRFE method*), 1005
save() (*evalml.pipelines.components.transformers.feature_selection.RFClassifierRFE method*), 1008
save() (*evalml.pipelines.components.transformers.feature_selection.recursive_feature_elimination_feature_selector.RFRegressorRFE method*), 1012
save() (*evalml.pipelines.components.transformers.feature_selection.RFRegressorRFE method*), 1015
save() (*evalml.pipelines.components.transformers.feature_selection.RFClassifierRFESelector method*), 1022
save() (*evalml.pipelines.components.transformers.feature_selection.RFClassifierSelectFromModel method*), 1025
save() (*evalml.pipelines.components.transformers.feature_selection.RFRegressorRFESelector method*), 1028
save() (*evalml.pipelines.components.transformers.feature_selection.RFRegressorSelectFromModel method*), 1031
save() (*evalml.pipelines.components.transformers.FeatureSelector method*), 1235
save() (*evalml.pipelines.components.transformers.Imputer method*), 1238
save() (*evalml.pipelines.components.transformers.imputers.Imputer method*), 1054
save() (*evalml.pipelines.components.transformers.imputers.imputer.Imputer method*), 1035
save() (*evalml.pipelines.components.transformers.imputers.knn_imputer.KNNImputer method*), 1038
save() (*evalml.pipelines.components.transformers.imputers.KNNImputer method*), 1057
save() (*evalml.pipelines.components.transformers.imputers.per_column_imputer method*), 1041
save() (*evalml.pipelines.components.transformers.imputers.PerColumnImputer method*), 1059
save() (*evalml.pipelines.components.transformers.imputers.simple_imputer method*), 1044
save() (*evalml.pipelines.components.transformers.imputers.SimpleImputer method*), 1062
save() (*evalml.pipelines.components.transformers.imputers.target_imputer method*), 1047
save() (*evalml.pipelines.components.transformers.imputers.TargetImputer method*), 1064
save() (*evalml.pipelines.components.transformers.imputers.time_series_imputer method*), 1051
save() (*evalml.pipelines.components.transformers.imputers.TimeSeriesImputer method*), 1067
save() (*evalml.pipelines.components.transformers.LabelEncoder method*), 1240
save() (*evalml.pipelines.components.transformers.LinearDiscriminantAnalysis method*), 1243
save() (*evalml.pipelines.components.transformers.LogTransformer method*), 1245
save() (*evalml.pipelines.components.transformers.LSA method*), 1248
save() (*evalml.pipelines.components.transformers.RecursiveFeatureElimination method*), 1250
save() (*evalml.pipelines.components.transformers.RFClassifierRFESelector method*), 1254
save() (*evalml.pipelines.components.transformers.RFRegressorRFESelector method*), 1257
save() (*evalml.pipelines.components.transformers.RFClassifierSelectFromModel method*), 1260
save() (*evalml.pipelines.components.transformers.RFRegressorSelectFromModel method*), 1262
save() (*evalml.pipelines.components.transformers.PerColumnImputer method*), 1265
save() (*evalml.pipelines.components.transformers.PolynomialDecomposer method*), 1270
save() (*evalml.pipelines.components.transformers.preprocessing.datetime_featurizer method*), 1070
save() (*evalml.pipelines.components.transformers.preprocessing.DateTimeFeaturizer method*), 1131

save() (evalml.pipelines.components.transformers.preprocessing.DateTime method), 1134
save() (evalml.pipelines.components.transformers.preprocessing.DelayedFeatureTransformer method), 1075
save() (evalml.pipelines.components.transformers.preprocessing.DFSTransformer method), 1138
save() (evalml.pipelines.components.transformers.preprocessing.DropNullColumns method), 1078
save() (evalml.pipelines.components.transformers.preprocessing.DropRowsWithNullOperator method), 1081
save() (evalml.pipelines.components.transformers.preprocessing.EmptyDataCheckPreprocessor method), 1084
save() (evalml.pipelines.components.transformers.preprocessing.EmptyNullOperator method), 1140
save() (evalml.pipelines.components.transformers.preprocessing.EmptyNullPipeline method), 1142
save() (evalml.pipelines.components.transformers.preprocessing.EmptyRowsTransformer method), 1145
save() (evalml.pipelines.components.transformers.preprocessing.EmptyTransformer method), 1147
save() (evalml.pipelines.components.transformers.preprocessing.FeatureSelectorTransformer method), 1087
save() (evalml.pipelines.components.transformers.preprocessing.LogTransformer method), 1090
save() (evalml.pipelines.components.transformers.preprocessing.LogTransformerPipeline method), 1150
save() (evalml.pipelines.components.transformers.preprocessing.LSA method), 1152
save() (evalml.pipelines.components.transformers.preprocessing.LSAComponent method), 1093
save() (evalml.pipelines.components.transformers.preprocessing.NaturalLanguageFeaturizer method), 1096
save() (evalml.pipelines.components.transformers.preprocessing.NaturalLanguageFeaturizerPipeline method), 1154
save() (evalml.pipelines.components.transformers.preprocessing.PolynomialDecomposer method), 1102
save() (evalml.pipelines.components.transformers.preprocessing.PolynomialDecomposerPipeline method), 1159
save() (evalml.pipelines.components.transformers.preprocessing.ReplaceNullableTypes method), 1105
save() (evalml.pipelines.components.transformers.preprocessing.ReplaceNullableTypesPipeline method), 1162
save() (evalml.pipelines.components.transformers.preprocessing.STLDecomposer method), 1111
save() (evalml.pipelines.components.transformers.preprocessing.STLDecomposerPipeline method), 1167
save() (evalml.pipelines.components.transformers.preprocessing.TextTransformer method), 1114
save() (evalml.pipelines.components.transformers.preprocessing.TextTransformerPipeline method), 1170
save() (evalml.pipelines.components.transformers.preprocessing.TimeSeriesFeaturizer method), 1118
save() (evalml.pipelines.components.transformers.preprocessing.TimeSeriesFeaturizerPipeline method), 1122
save() (evalml.pipelines.components.transformers.preprocessing.TimeSeriesImputer method), 1173
save() (evalml.pipelines.components.transformers.preprocessing.TimeSeriesImputerPipeline method), 1176
save() (evalml.pipelines.components.transformers.preprocessing.transformer.Transformer method), 1124
save() (evalml.pipelines.components.transformers.preprocessing.transformer.TransformerPipeline method), 1127
save() (evalml.pipelines.components.transformers.preprocessing.URLFeaturizer method), 1179
save() (evalml.pipelines.components.transformers.transformer.ReplaceNullableTypes method), 1272
save() (evalml.pipelines.components.transformers.RFClassifierRFESelector method), 1275
save() (evalml.pipelines.components.transformers.RFClassifierSelectFromModel method), 1278
save() (evalml.pipelines.components.transformers.RFRegressorRFESelector method), 1281
save() (evalml.pipelines.components.transformers.RFRegressorSelectFromModel method), 1284
save() (evalml.pipelines.components.transformers.samplers.base_sampler.BaseSampler method), 1182
save() (evalml.pipelines.components.transformers.samplers.Oversampler method), 1191
save() (evalml.pipelines.components.transformers.samplers.oversampler.Oversampler method), 1185
save() (evalml.pipelines.components.transformers.samplers.Undersampler method), 1195
save() (evalml.pipelines.components.transformers.samplers.undersampler.Undersampler method), 1189
save() (evalml.pipelines.components.transformers.scalers.standard_scaler.StandardScaler method), 1197
save() (evalml.pipelines.components.transformers.scalers.StandardScaler method), 1200
save() (evalml.pipelines.components.transformer.SelectByType method), 1287
save() (evalml.pipelines.components.transformers.SelectColumns method), 1289
save() (evalml.pipelines.components.transformers.SimpleImputer method), 1292
save() (evalml.pipelines.components.transformers.StandardScaler method), 1294
save() (evalml.pipelines.components.transformers.STLDecomposer method), 1299
save() (evalml.pipelines.components.transformers.TargetEncoder method), 1302
save() (evalml.pipelines.components.transformers.TargetImputer method), 1305
save() (evalml.pipelines.components.transformers.TimeSeriesFeaturizer method), 1308
save() (evalml.pipelines.components.transformers.TimeSeriesImputer method), 1311
save() (evalml.pipelines.components.transformers.TimeSeriesRegularizer method), 1314

save() (*evalml.pipelines.components.transformers.Transformer method*), 1317
save() (*evalml.pipelines.components.transformers.transformer.Transformer method*), 1213
save() (*evalml.pipelines.components.transformers.Undersampler method*), 1320
save() (*evalml.pipelines.components.transformers.URLFeaturizer method*), 1323
save() (*evalml.pipelines.components.Undersampler method*), 1534
save() (*evalml.pipelines.components.URLFeaturizer method*), 1536
save() (*evalml.pipelines.components.VowpalWabbitBinaryClassifier method*), 1540
save() (*evalml.pipelines.components.VowpalWabbitMulticlassClassifier method*), 1543
save() (*evalml.pipelines.components.VowpalWabbitRegressor method*), 1546
save() (*evalml.pipelines.components.XGBoostClassifier method*), 1549
save() (*evalml.pipelines.components.XGBoostRegressor method*), 1552
save() (*evalml.pipelines.DecisionTreeClassifier method*), 1674
save() (*evalml.pipelines.DecisionTreeRegressor method*), 1678
save() (*evalml.pipelines.DFSTransformer method*), 1680
save() (*evalml.pipelines.DropNaNRowsTransformer method*), 1683
save() (*evalml.pipelines.ElasticNetClassifier method*), 1687
save() (*evalml.pipelines.ElasticNetRegressor method*), 1690
save() (*evalml.pipelines.Estimator method*), 1693
save() (*evalml.pipelines.ExponentialSmoothingRegressor method*), 1696
save() (*evalml.pipelines.ExtraTreesClassifier method*), 1700
save() (*evalml.pipelines.ExtraTreesRegressor method*), 1704
save() (*evalml.pipelines.FeatureSelector method*), 1706
save() (*evalml.pipelines.Imputer method*), 1709
save() (*evalml.pipelines.KNeighborsClassifier method*), 1713
save() (*evalml.pipelines.LightGBMClassifier method*), 1716
save() (*evalml.pipelines.LightGBMRegressor method*), 1720
save() (*evalml.pipelines.LinearRegressor method*), 1723
save() (*evalml.pipelines.LogisticRegressionClassifier method*), 1726
save() (*evalml.pipelines.multiclass_classification_pipeline.MulticlassClassificationPipeline method*), 1580
save() (*evalml.pipelines.MulticlassClassificationPipeline method*), 1732
save() (*evalml.pipelines.OneHotEncoder method*), 1736
save() (*evalml.pipelines.OrdinalEncoder method*), 1739
save() (*evalml.pipelines.PerColumnImputer method*), 1741
save() (*evalml.pipelines.pipeline_base.PipelineBase method*), 1586
save() (*evalml.pipelines.PipelineBase method*), 1746
save() (*evalml.pipelines.ProphetRegressor method*), 1751
save() (*evalml.pipelines.RandomForestClassifier method*), 1754
save() (*evalml.pipelines.RandomForestRegressor method*), 1757
save() (*evalml.pipelines.regression_pipeline.RegressionPipeline method*), 1594
save() (*evalml.pipelines.RegressionPipeline method*), 1762
save() (*evalml.pipelines.RFClassifierSelectFromModel method*), 1766
save() (*evalml.pipelines.RFRegressorSelectFromModel method*), 1769
save() (*evalml.pipelines.SimpleImputer method*), 1771
save() (*evalml.pipelines.StackedEnsembleBase method*), 1775
save() (*evalml.pipelines.StackedEnsembleClassifier method*), 1779
save() (*evalml.pipelines.StackedEnsembleRegressor method*), 1782
save() (*evalml.pipelines.StandardScaler method*), 1784
save() (*evalml.pipelines.SVMClassifier method*), 1788
save() (*evalml.pipelines.SVMRegressor method*), 1791
save() (*evalml.pipelines.TargetEncoder method*), 1793
save() (*evalml.pipelines.time_series_classification_pipelines.TimeSeriesB method*), 1603
save() (*evalml.pipelines.time_series_classification_pipelines.TimeSeriesC method*), 1610
save() (*evalml.pipelines.time_series_classification_pipelines.TimeSeriesM method*), 1618
save() (*evalml.pipelines.time_series_pipeline_base.TimeSeriesPipelineBa method*), 1625
save() (*evalml.pipelines.time_series_regression_pipeline.TimeSeriesRegre method*), 1634
save() (*evalml.pipelines.TimeSeriesBinaryClassificationPipeline method*), 1801
save() (*evalml.pipelines.TimeSeriesClassificationPipeline method*), 1808
save() (*evalml.pipelines.TimeSeriesFeaturizer method*), 1811
save() (*evalml.pipelines.TimeSeriesImputer method*), 1814
save() (*evalml.pipelines.TimeSeriesMulticlassClassificationPipeline method*), 1821

save() (*evalml.pipelines.TimeSeriesRegressionPipeline method*), 1830
save() (*evalml.pipelines.TimeSeriesRegularizer method*), 1834
save() (*evalml.pipelines.Transformer method*), 1836
save() (*evalml.pipelines.VowpalWabbitBinaryClassifier method*), 1840
save() (*evalml.pipelines.VowpalWabbitMulticlassClassifier method*), 1843
save() (*evalml.pipelines.VowpalWabbitRegressor method*), 1846
save() (*evalml.pipelines.XGBoostClassifier method*), 1849
save() (*evalml.pipelines.XGBoostRegressor method*), 1852
save_plot() (*in module evalml.utils*), 1913
save_plot() (*in module evalml.utils.gen_utils*), 1905
scikit_learn_wrapped_estimator() (*in module evalml.pipelines.components.utils*), 1331
score() (*evalml.objectives.AccuracyBinary method*), 536
score() (*evalml.objectives.AccuracyMulticlass method*), 537
score() (*evalml.objectives.AUC method*), 540
score() (*evalml.objectives.AUCMacro method*), 541
score() (*evalml.objectives.AUCMicro method*), 543
score() (*evalml.objectives.AUCWeighted method*), 545
score() (*evalml.objectives.BalancedAccuracyBinary method*), 547
score() (*evalml.objectives.BalancedAccuracyMulticlass method*), 549
score() (*evalml.objectives.binary_classification_objective.BinaryClassificationObjective method*), 443
score() (*evalml.objectives.BinaryClassificationObjective method*), 552
score() (*evalml.objectives.cost_benefit_matrix.CostBenefitMatrix method*), 446
score() (*evalml.objectives.CostBenefitMatrix method*), 554
score() (*evalml.objectives.ExpVariance method*), 556
score() (*evalml.objectives.F1 method*), 558
score() (*evalml.objectives.F1Macro method*), 559
score() (*evalml.objectives.F1Micro method*), 561
score() (*evalml.objectives.F1Weighted method*), 563
score() (*evalml.objectives.fraud_cost.FraudCost method*), 448
score() (*evalml.objectives.FraudCost method*), 565
score() (*evalml.objectives.Gini method*), 569
score() (*evalml.objectives.lead_scoring.LeadScoring method*), 451
score() (*evalml.objectives.LeadScoring method*), 571
score() (*evalml.objectives.LogLossBinary method*), 574
score() (*evalml.objectives.LogLossMulticlass method*), 575
score() (*evalml.objectives.MAE method*), 577
score() (*evalml.objectives.MAPE method*), 579
score() (*evalml.objectives.MaxError method*), 581
score() (*evalml.objectives.MCCBinary method*), 583
score() (*evalml.objectives.MCCMulticlass method*), 584
score() (*evalml.objectives.MeanSquaredLogError method*), 586
score() (*evalml.objectives.MedianAE method*), 588
score() (*evalml.objectives.MSE method*), 589
score() (*evalml.objectives.multiclass_classification_objective.MulticlassC method*), 453
score() (*evalml.objectives.MulticlassClassificationObjective method*), 591
score() (*evalml.objectives.objective_base.ObjectiveBase method*), 456
score() (*evalml.objectives.ObjectiveBase method*), 594
score() (*evalml.objectives.Precision method*), 596
score() (*evalml.objectives.PrecisionMacro method*), 598
score() (*evalml.objectives.PrecisionMicro method*), 599
score() (*evalml.objectives.PrecisionWeighted method*), 601
score() (*evalml.objectives.R2 method*), 603
score() (*evalml.objectives.Recall method*), 605
score() (*evalml.objectives.RecallMacro method*), 607
score() (*evalml.objectives.RecallMicro method*), 608
score() (*evalml.objectives.RecallWeighted method*), 610
score() (*evalml.objectives.regression_objective.RegressionObjective method*), 458
score() (*evalml.objectives.RegressionObjective method*), 612
score() (*evalml.objectives.RootMeanSquaredError method*), 614
score() (*evalml.objectives.RootMeanSquaredLogError method*), 616
score() (*evalml.objectives.sensitivity_low_alert.SensitivityLowAlert method*), 461
score() (*evalml.objectives.SensitivityLowAlert method*), 618
score() (*evalml.objectives.standard_metrics.AccuracyBinary method*), 465
score() (*evalml.objectives.standard_metrics.AccuracyMulticlass method*), 466
score() (*evalml.objectives.standard_metrics.AUC method*), 469
score() (*evalml.objectives.standard_metrics.AUCMacro method*), 470
score() (*evalml.objectives.standard_metrics.AUCMicro method*), 472
score() (*evalml.objectives.standard_metrics.AUCWeighted method*), 474
score() (*evalml.objectives.standard_metrics.BalancedAccuracyBinary method*), 476

score() (*evalml.objectives.standard_metrics.BalancedAccuracyMulticlass method*), 478
score() (*evalml.objectives.standard_metrics.ExpVariance method*), 479
score() (*evalml.objectives.standard_metrics.F1 method*), 482
score() (*evalml.objectives.standard_metrics.F1Macro method*), 483
score() (*evalml.objectives.standard_metrics.F1Micro method*), 485
score() (*evalml.objectives.standard_metrics.F1Weighted method*), 487
score() (*evalml.objectives.standard_metrics.Gini method*), 489
score() (*evalml.objectives.standard_metrics.LogLossBinary method*), 491
score() (*evalml.objectives.standard_metrics.LogLossMulticlass method*), 493
score() (*evalml.objectives.standard_metrics.MAE method*), 495
score() (*evalml.objectives.standard_metrics.MAPE method*), 496
score() (*evalml.objectives.standard_metrics.MaxError method*), 498
score() (*evalml.objectives.standard_metrics.MCCBinary method*), 500
score() (*evalml.objectives.standard_metrics.MCCMulticlass method*), 502
score() (*evalml.objectives.standard_metrics.MeanSquaredLogError method*), 504
score() (*evalml.objectives.standard_metrics.MedianAE method*), 506
score() (*evalml.objectives.standard_metrics.MSE method*), 507
score() (*evalml.objectives.standard_metrics.Precision method*), 509
score() (*evalml.objectives.standard_metrics.PrecisionMacro method*), 511
score() (*evalml.objectives.standard_metrics.PrecisionMicro method*), 513
score() (*evalml.objectives.standard_metrics.PrecisionWeighted method*), 514
score() (*evalml.objectives.standard_metrics.R2 method*), 516
score() (*evalml.objectives.standard_metrics.Recall method*), 518
score() (*evalml.objectives.standard_metrics.RecallMacro method*), 520
score() (*evalml.objectives.standard_metrics.RecallMicro method*), 522
score() (*evalml.objectives.standard_metrics.RecallWeighted method*), 523
score() (*evalml.objectives.standard_metrics.RootMeanSquaredError method*), 525
score() (*evalml.objectives.standard_metrics.RootMeanSquaredLogError method*), 527
score() (*evalml.objectives.time_series_regression_objective.TimeSeriesRe method*), 529
score() (*evalml.pipelines.binary_classification_pipeline.BinaryClassificat method*), 1559
score() (*evalml.pipelines.BinaryClassificationPipeline method*), 1651
score() (*evalml.pipelines.classification_pipeline.ClassificationPipeline method*), 1567
score() (*evalml.pipelines.ClassificationPipeline method*), 1664
score() (*evalml.pipelines.components.utils.WrappedSKClassifier method*), 1332
score() (*evalml.pipelines.components.utils.WrappedSKRegressor method*), 1333
score() (*evalml.pipelines.multiclass_classification_pipeline.MulticlassCla method*), 1580
score() (*evalml.pipelines.MulticlassClassificationPipeline method*), 1732
score() (*evalml.pipelines.pipeline_base.PipelineBase method*), 1586
score() (*evalml.pipelines.PipelineBase method*), 1746
score() (*evalml.pipelines.regression_pipeline.RegressionPipeline method*), 1594
score() (*evalml.pipelines.RegressionPipeline method*), 1762
score() (*evalml.pipelines.time_series_classification_pipelines.TimeSeries method*), 1603
score() (*evalml.pipelines.time_series_classification_pipelines.TimeSeries method*), 1610
score() (*evalml.pipelines.time_series_classification_pipelines.TimeSeries method*), 1618
score() (*evalml.pipelines.time_series_pipeline_base.TimeSeriesPipelineB method*), 1625
score() (*evalml.pipelines.time_series_regression_pipeline.TimeSeriesReg method*), 1634
score() (*evalml.pipelines.TimeSeriesBinaryClassificationPipeline method*), 1801
score() (*evalml.pipelines.TimeSeriesClassificationPipeline method*), 1808
score() (*evalml.pipelines.TimeSeriesMulticlassClassificationPipeline method*), 1821
score() (*evalml.pipelines.TimeSeriesRegressionPipeline method*), 1830
score_needs_proba (*evalml.objectives.binary_classification_objective.Bi property*), 443
score_needs_proba (*evalml.objectives.BinaryClassificationObjective property*), 552
score_needs_proba (*evalml.objectives.multiclass_classification_objective property*), 454
score_needs_proba (*evalml.objectives.MulticlassClassificationObjective property*), 592
score_needs_proba (*evalml.objectives.objective_base.ObjectiveBase*

*property*), 456
score_needs_proba (*evalml.objectives.ObjectiveBase property*), 594
score_needs_proba (*evalml.objectives.regression_objectives.RegressionObjective property*), 459
score_needs_proba (*evalml.objectives.RegressionObjective property*), 613
score_needs_proba (*evalml.objectives.time_series_regression_objectives.TimeSeriesRegressionObjective property*), 530
score_pipeline() (*in module evalml.automl.engine.engine_base*), 246
score_pipelines() (*evalml.automl.automl_search.AutoMLSearch method*), 263
score_pipelines() (*evalml.automl.AutoMLSearch method*), 279
score_pipelines() (*evalml.AutoMLSearch method*), 1919
search() (*evalml.automl.automl_search.AutoMLSearch method*), 263
search() (*evalml.automl.AutoMLSearch method*), 279
search() (*evalml.AutoMLSearch method*), 1919
search() (*in module evalml*), 1920
search() (*in module evalml.automl*), 282
search() (*in module evalml.automl.automl_search*), 264
search_iteration_plot() (*evalml.automl.pipeline_search_plots.PipelineSearchPlots method*), 267
search_iterative() (*in module evalml*), 1921
search_iterative() (*in module evalml.automl*), 283
search_iterative() (*in module evalml.automl.automl_search*), 265
SearchIterationPlot (*class in evalml.automl.pipeline_search_plots*), 267
SEED_BOUNDS (*in module evalml.utils*), 1913
SEED_BOUNDS (*in module evalml.utils.gen_utils*), 1905
SelectByType (*class in evalml.pipelines.components*), 1480
SelectByType (*class in evalml.pipelines.components.transformers*), 1285
SelectByType (*class in evalml.pipelines.components.transformers.column_selectors*), 1206
SelectColumns (*class in evalml.pipelines.components*), 1482
SelectColumns (*class in evalml.pipelines.components.transformers*), 1287
SelectColumns (*class in evalml.pipelines.components.transformers.column_selectors*), 1208
send_data_to_cluster() (*evalml.automl.engine.dask_engine.DaskEngine method*), 242
send_data_to_cluster() (*evalml.automl.engine.DaskEngine method*), 252
SensitivityLowAlert (*class in evalml.objectives*), 616
SensitivityLowAlert (*class in evalml.objectives.sensitivity_low_alert*), 459
SequentialComputation (*class in evalml.automl.engine.sequential_engine*), 248
SequentialEngine (*class in evalml.automl*), 284
SequentialEngine (*class in evalml.automl.engine*), 255
SequentialEngine (*class in evalml.automl.engine.sequential_engine*), 249
set_fit() (*evalml.pipelines.components.component_base_meta.Compone class method*), 1327
set_fit() (*evalml.pipelines.components.ComponentBaseMeta class method*), 1356
set_fit() (*evalml.pipelines.components.transformers.encoders.onehot_en class method*), 975
set_fit() (*evalml.pipelines.components.transformers.encoders.ordinal_e class method*), 980
set_fit() (*evalml.pipelines.components.transformers.imputers.target_imp class method*), 1048
set_fit() (*evalml.pipelines.pipeline_meta.PipelineBaseMeta class method*), 1588
set_fit() (*evalml.utils.base_meta.BaseMeta class method*), 1898
set_params() (*evalml.pipelines.components.utils.WrappedSKClassifier method*), 1332
set_params() (*evalml.pipelines.components.utils.WrappedSKRegressor method*), 1334
set_period() (*evalml.pipelines.components.PolynomialDecomposer method*), 1455
set_period() (*evalml.pipelines.components.STLDecomposer method*), 1504
set_period() (*evalml.pipelines.components.transformers.PolynomialDec method*), 1270
set_period() (*evalml.pipelines.components.transformers.preprocessing.L method*), 1134
set_period() (*evalml.pipelines.components.transformers.preprocessing.d method*), 1075
set_period() (*evalml.pipelines.components.transformers.preprocessing.p method*), 1102
set_period() (*evalml.pipelines.components.transformers.preprocessing.F method*), 1159
set_period() (*evalml.pipelines.components.transformers.preprocessing.s method*), 1111
set_period() (*evalml.pipelines.components.transformers.preprocessing.S method*), 1167
set_period() (*evalml.pipelines.components.transformers.STLDecompose method*), 1299
setup_job_log() (*evalml.automl.engine.cf_engine.CFEngine static method*), 240

setup_job_log() (*evalml.automl.engine.CFEngine static method*), 251
setup_job_log() (*evalml.automl.engine.dask_engine.DaskEngine static method*), 243
setup_job_log() (*evalml.automl.engine.DaskEngine static method*), 252
setup_job_log() (*evalml.automl.engine.engine_base.EngineBase static method*), 245
setup_job_log() (*evalml.automl.engine.EngineBase static method*), 254
setup_job_log() (*evalml.automl.engine.sequential_engine.SequentialEngine static method*), 249
setup_job_log() (*evalml.automl.engine.SequentialEngine static method*), 255
setup_job_log() (*evalml.automl.EngineBase static method*), 280
setup_job_log() (*evalml.automl.SequentialEngine static method*), 284
should_continue() (*evalml.automl.Progress method*), 281
should_continue() (*evalml.automl.progress.Progress method*), 269
silent_error_callback() (*in module evalml.automl.callbacks*), 266
SimpleImputer (*class in evalml.pipelines*), 1769
SimpleImputer (*class in evalml.pipelines.components*), 1484
SimpleImputer (*class in evalml.pipelines.components.transformers*), 1289
SimpleImputer (*class in evalml.pipelines.components.transformers.imputers*), 1059
SimpleImputer (*class in evalml.pipelines.components.transformers.imputers.simple_imputer*), 1042
SKOptTuner (*class in evalml.tuners*), 1895
SKOptTuner (*class in evalml.tuners.skopt_tuner*), 1888
sparsity_score() (*evalml.data_checks.sparsity_data_check.SparsityDataCheck static method*), 328
sparsity_score() (*evalml.data_checks.SparsityDataCheck static method*), 378
SparsityDataCheck (*class in evalml.data_checks*), 377
SparsityDataCheck (*class in evalml.data_checks.sparsity_data_check*), 328
split() (*evalml.preprocessing.data_splitters.KFold method*), 1861
split() (*evalml.preprocessing.data_splitters.no_split.NoSplit method*), 1853
split() (*evalml.preprocessing.data_splitters.NoSplit method*), 1861
split() (*evalml.preprocessing.data_splitters.sk_splitters.KFold method*), 1854
split() (*evalml.preprocessing.data_splitters.sk_splitters.StratifiedKFold method*), 1855
split() (*evalml.preprocessing.data_splitters.StratifiedKFold method*), 1862
split() (*evalml.preprocessing.data_splitters.time_series_split.TimeSeries method*), 1858
split() (*evalml.preprocessing.data_splitters.TimeSeriesSplit method*), 1864
split() (*evalml.preprocessing.data_splitters.training_validation_split.Tra method*), 1860
split() (*evalml.preprocessing.data_splitters.TrainingValidationSplit method*), 1866
split() (*evalml.preprocessing.NoSplit method*), 1870
split() (*evalml.preprocessing.TimeSeriesSplit method*), 1873
split() (*evalml.preprocessing.TrainingValidationSplit method*), 1875
split_data() (*in module evalml.preprocessing*), 1870
split_data() (*in module evalml.preprocessing.utils*), 1867
StackedEnsembleBase (*class in evalml.pipelines*), 1772
StackedEnsembleBase (*class in evalml.pipelines.components*), 1487
StackedEnsembleBase (*class in evalml.pipelines.components.ensemble*), 631
StackedEnsembleBase (*class in evalml.pipelines.components.ensemble.stacked_ensemble_base*), 619
StackedEnsembleClassifier (*class in evalml.pipelines*), 1775
StackedEnsembleClassifier (*class in evalml.pipelines.components*), 1490
StackedEnsembleClassifier (*class in evalml.pipelines.components.ensemble*), 634
StackedEnsembleClassifier (*class in evalml.pipelines.components.ensemble.stacked_ensemble_classifi 622*
StackedEnsembleRegressor (*class in evalml.pipelines*), 1779
StackedEnsembleRegressor (*class in evalml.pipelines.components*), 1494
StackedEnsembleRegressor (*class in evalml.pipelines.components.ensemble*), 638
StackedEnsembleRegressor (*class in evalml.pipelines.components.ensemble.stacked_ensemble_regress 627*
standardize_format() (*in module evalml.utils.cli_utils*), 1900
StandardScaler (*class in evalml.pipelines*), 1782
StandardScaler (*class in evalml.pipelines.components*), 1497

StandardScaler (*class in evalml.pipelines.components.transformers*), 1292
StandardScaler (*class in evalml.pipelines.components.transformers.scalers*), 1198
StandardScaler (*class in evalml.pipelines.components.transformers.scalers.standard_scaler*), 1196
start_timing() (*evalml.automl.Progress method*), 282
start_timing() (*evalml.automl.progress.Progress method*), 269
STLDecomposer (*class in evalml.pipelines.components*), 1500
STLDecomposer (*class in evalml.pipelines.components.transformers*), 1294
STLDecomposer (*class in evalml.pipelines.components.transformers.preprocessing*), 1163
STLDecomposer (*class in evalml.pipelines.components.transformers.preprocessing.stl_decomposer*), 1106
StratifiedKFold (*class in evalml.preprocessing.data_splitters*), 1862
StratifiedKFold (*class in evalml.preprocessing.data_splitters.sk_splitters*), 1855
submit() (*evalml.automl.engine.cf_engine.CFClient method*), 239
submit_evaluation_job() (*evalml.automl.engine.cf_engine.CFEngine method*), 240
submit_evaluation_job() (*evalml.automl.engine.CFEngine method*), 251
submit_evaluation_job() (*evalml.automl.engine.dask_engine.DaskEngine method*), 243
submit_evaluation_job() (*evalml.automl.engine.DaskEngine method*), 253
submit_evaluation_job() (*evalml.automl.engine.engine_base.EngineBase method*), 245
submit_evaluation_job() (*evalml.automl.engine.EngineBase method*), 254
submit_evaluation_job() (*evalml.automl.engine.sequential_engine.SequentialEngine method*), 249
submit_evaluation_job() (*evalml.automl.engine.SequentialEngine method*), 255
submit_evaluation_job() (*evalml.automl.EngineBase method*), 280
submit_evaluation_job() (*evalml.automl.SequentialEngine method*), 284
submit_scoring_job() (*evalml.automl.engine.cf_engine.CFEngine method*), 240
submit_scoring_job() (*evalml.automl.engine.CFEngine method*), 251
submit_scoring_job() (*evalml.automl.engine.dask_engine.DaskEngine method*), 243
submit_scoring_job() (*evalml.automl.engine.DaskEngine method*), 253
submit_scoring_job() (*evalml.automl.engine.engine_base.EngineBase method*), 245
submit_scoring_job() (*evalml.automl.engine.EngineBase method*), 254
submit_scoring_job() (*evalml.automl.engine.sequential_engine.SequentialEngine method*), 249
submit_scoring_job() (*evalml.automl.engine.SequentialEngine method*), 255
submit_scoring_job() (*evalml.automl.EngineBase method*), 280
submit_scoring_job() (*evalml.automl.SequentialEngine method*), 284
submit_training_job() (*evalml.automl.engine.cf_engine.CFEngine method*), 241
submit_training_job() (*evalml.automl.engine.CFEngine method*), 252
submit_training_job() (*evalml.automl.engine.dask_engine.DaskEngine method*), 243
submit_training_job() (*evalml.automl.engine.DaskEngine method*), 253
submit_training_job() (*evalml.automl.engine.engine_base.EngineBase method*), 245
submit_training_job() (*evalml.automl.engine.EngineBase method*), 254
submit_training_job() (*evalml.automl.engine.sequential_engine.SequentialEngine*

*method*), 249
submit_training_job() (*evalml.automl.engine.SequentialEngine method*), 256
submit_training_job() (*evalml.automl.EngineBase method*), 280
submit_training_job() (*evalml.automl.SequentialEngine method*), 285
summary (*evalml.pipelines.binary_classification_pipeline.BinaryClassificationPipeline property*), 1560
summary (*evalml.pipelines.BinaryClassificationPipeline property*), 1652
summary (*evalml.pipelines.classification_pipeline.ClassificationPipeline property*), 1567
summary (*evalml.pipelines.ClassificationPipeline property*), 1664
summary (*evalml.pipelines.multiclass_classification_pipeline.MulticlassClassificationPipeline property*), 1580
summary (*evalml.pipelines.MulticlassClassificationPipeline property*), 1732
summary (*evalml.pipelines.pipeline_base.PipelineBase property*), 1587
summary (*evalml.pipelines.PipelineBase property*), 1747
summary (*evalml.pipelines.regression_pipeline.RegressionPipeline property*), 1595
summary (*evalml.pipelines.RegressionPipeline property*), 1763
summary (*evalml.pipelines.time_series_classification_pipelines.TimeSeriesBinaryClassificationPipeline property*), 1603
summary (*evalml.pipelines.time_series_classification_pipelines.TimeSeriesClassificationPipeline property*), 1611
summary (*evalml.pipelines.time_series_classification_pipelines.TimeSeriesMulticlassClassificationPipeline property*), 1618
summary (*evalml.pipelines.time_series_pipeline_base.TimeSeriesPipelineBase property*), 1626
summary (*evalml.pipelines.time_series_regression_pipeline.TimeSeriesRegressionPipeline property*), 1635
summary (*evalml.pipelines.TimeSeriesBinaryClassificationPipeline property*), 1801
summary (*evalml.pipelines.TimeSeriesClassificationPipeline property*), 1808
summary (*evalml.pipelines.TimeSeriesMulticlassClassificationPipeline property*), 1822
summary (*evalml.pipelines.TimeSeriesRegressionPipeline property*), 1830
supported_problem_types (*evalml.pipelines.components.ensemble.stacked_ensemble_base.StackedEnsembleBase property*), 622
supported_problem_types (*evalml.pipelines.components.ensemble.StackedEnsembleBase property*), 634
supported_problem_types (*evalml.pipelines.components.Estimator property*), 1391
supported_problem_types (*evalml.pipelines.components.estimators.classifiers.vowpal_wabbit property*), 685
supported_problem_types (*evalml.pipelines.components.estimators.Estimator property*), 893
supported_problem_types (*evalml.pipelines.components.estimators.estimator.Estimator property*), 857
supported_problem_types (*evalml.pipelines.components.StackedEnsembleBase property*), 1490
supported_problem_types (*evalml.pipelines.Estimator property*), 1693
supported_problem_types (*evalml.pipelines.StackedEnsembleBase property*), 1775
SVMClassifier (*class in evalml.pipelines*), 1785
SVMClassifier (*class in evalml.pipelines.components*), 1505
SVMClassifier (*class in evalml.pipelines.components.estimators*), 931
SVMClassifier (*class in evalml.pipelines.components.estimators.classifiers*), 727
SVMClassifier (*class in evalml.pipelines.components.estimators.classifiers.svm_classifier*), 678
SVMRegressor (*class in evalml.pipelines*), 1788
SVMRegressor (*class in evalml.pipelines.components*), 1508
SVMRegressor (*class in evalml.pipelines.components.estimators*), 934
SVMRegressor (*class in evalml.pipelines.components.estimators.regressors*), 840
SVMRegressor (*class in evalml.pipelines.components.estimators.regressors.svm_regressor*), 786


## T


t_sne() (*in module evalml.model_understanding*), 439
t_sne() (*in module evalml.model_understanding.visualizations*), 425
target_distribution() (*in module evalml.preprocessing*), 1871
target_distribution() (*in module evalml.preprocessing.utils*), 1868
TargetDistributionDataCheck (*class in evalml.data_checks*), 379
TargetDistributionDataCheck (*class in*

*evalml.data_checks.target_distribution_data_check*), 330
TargetEncoder (*class in evalml.pipelines*), 1791
TargetEncoder (*class in evalml.pipelines.components*), 1511
TargetEncoder (*class in evalml.pipelines.components.transformers*), 1300
TargetEncoder (*class in evalml.pipelines.components.transformers.encoders*), 993
TargetEncoder (*class in evalml.pipelines.components.transformers.encoders.target_encoder*), 981
TargetImputer (*class in evalml.pipelines.components*), 1514
TargetImputer (*class in evalml.pipelines.components.transformers*), 1303
TargetImputer (*class in evalml.pipelines.components.transformers.imputers*), 1062
TargetImputer (*class in evalml.pipelines.components.transformers.imputers.target_imputer*), 1045
TargetImputerMeta (*class in evalml.pipelines.components.transformers.imputers.target_imputer*), 1047
TargetLeakageDataCheck (*class in evalml.data_checks*), 380
TargetLeakageDataCheck (*class in evalml.data_checks.target_leakage_data_check*), 332
TextTransformer (*class in evalml.pipelines.components.transformers.preprocessing*), 1168
TextTransformer (*class in evalml.pipelines.components.transformers.preprocessing.text_transformer*), 1112
threshold (*evalml.pipelines.binary_classification_pipeline.BinaryClassificationPipelineMixin property*), 1560
threshold (*evalml.pipelines.binary_classification_pipeline_mixin.BinaryClassificationPipelineMixin property*), 1561
threshold (*evalml.pipelines.BinaryClassificationPipeline property*), 1652
threshold (*evalml.pipelines.time_series_classification_pipelines.TimeSeriesBinaryClassificationPipeline property*), 1603
threshold (*evalml.pipelines.TimeSeriesBinaryClassificationPipeline property*), 1801
time_elapsed() (*in module evalml.utils.logger*), 1906
TimeSeriesBaselineEstimator (*class in evalml.pipelines.components*), 1517
TimeSeriesBaselineEstimator (*class in evalml.pipelines.components.estimators*), 937
TimeSeriesBaselineEstimator (*class in evalml.pipelines.components.estimators.regressors*), 843
TimeSeriesBaselineEstimator (*class in evalml.pipelines.components.estimators.regressors.time_series_b*
790
TimeSeriesBinaryClassificationPipeline (*class in evalml.pipelines*), 1794
TimeSeriesBinaryClassificationPipeline (*class in evalml.pipelines.time_series_classification_pipelines*), 1596
TimeSeriesClassificationPipeline (*class in evalml.pipelines*), 1802
TimeSeriesClassificationPipeline (*class in evalml.pipelines.time_series_classification_pipelines*), 1604
TimeSeriesFeaturizer (*class in evalml.pipelines*), 1809
TimeSeriesFeaturizer (*class in evalml.pipelines.components*), 1520
TimeSeriesFeaturizer (*class in evalml.pipelines.components.transformers*), 1305
TimeSeriesFeaturizer (*class in evalml.pipelines.components.transformers.preprocessing*), 1171
TimeSeriesFeaturizer (*class in evalml.pipelines.components.transformers.preprocessing.time_se*
1115
TimeSeriesImputer (*class in evalml.pipelines*), 1812
TimeSeriesImputer (*class in evalml.pipelines.components*), 1523
TimeSeriesImputer (*class in evalml.pipelines.components.transformers*), 1309
TimeSeriesImputer (*class in evalml.pipelines.components.transformers.imputers*), 1065
TimeSeriesImputerPipeline (*class in evalml.pipelines.components.transformers.imputers.time_series_i*
1049
TimeSeriesMulticlassClassificationPipeline (*class in evalml.pipelines*), 1815
TimeSeriesMulticlassClassificationPipeline (*class in evalml.pipelines.time_series_classification_pipelines*), 1611
TimeSeriesParametersDataCheck (*class in evalml.data_checks*), 382
TimeSeriesParametersDataCheck (*class in evalml.data_checks.ts_parameters_data_check*), 334
TimeSeriesPipelineBase (*class in evalml.pipelines.time_series_pipeline_base*),

1619
TimeSeriesRegressionObjective (*class in evalml.objectives.time_series_regression_objective*), 528
TimeSeriesRegressionPipeline (*class in evalml.pipelines*), 1822
TimeSeriesRegressionPipeline (*class in evalml.pipelines.time_series_regression_pipeline*), 1627
TimeSeriesRegularizer (*class in evalml.pipelines*), 1831
TimeSeriesRegularizer (*class in evalml.pipelines.components*), 1526
TimeSeriesRegularizer (*class in evalml.pipelines.components.transformers*), 1312
TimeSeriesRegularizer (*class in evalml.pipelines.components.transformers.preprocessing*), 1174
TimeSeriesRegularizer (*class in evalml.pipelines.components.transformers.preprocessing.time_series_preprocessing*), 1119
TimeSeriesSplit (*class in evalml.preprocessing*), 1872
TimeSeriesSplit (*class in evalml.preprocessing.data_splitters*), 1863
TimeSeriesSplit (*class in evalml.preprocessing.data_splitters.time_series_split*), 1856
TimeSeriesSplittingDataCheck (*class in evalml.data_checks*), 384
TimeSeriesSplittingDataCheck (*class in evalml.data_checks.ts_splitting_data_check*), 336
to_dict() (*evalml.data_checks.data_check_action.DataCheckAction method*), 290
to_dict() (*evalml.data_checks.data_check_action_option.DataCheckActionOption method*), 293
to_dict() (*evalml.data_checks.data_check_message.DataCheckError method*), 295
to_dict() (*evalml.data_checks.data_check_message.DataCheckMessage method*), 295
to_dict() (*evalml.data_checks.data_check_message.DataCheckWarning method*), 296
to_dict() (*evalml.data_checks.DataCheckAction method*), 346
to_dict() (*evalml.data_checks.DataCheckActionOption method*), 348
to_dict() (*evalml.data_checks.DataCheckError method*), 348
to_dict() (*evalml.data_checks.DataCheckMessage method*), 348
to_dict() (*evalml.data_checks.DataCheckWarning method*), 351
train_and_score_pipeline() (*in module evalml.automl.engine*), 256
train_and_score_pipeline() (*in module evalml.automl.engine.engine_base*), 247
train_pipeline() (*in module evalml.automl.engine*), 257
train_pipeline() (*in module evalml.automl.engine.engine_base*), 247
train_pipelines() (*evalml.automl.automl_search.AutoMLSearch method*), 263
train_pipelines() (*evalml.automl.AutoMLSearch method*), 279
train_pipelines() (*evalml.AutoMLSearch method*), 1919
training_only (*evalml.pipelines.components.component_base.Component property*), 1326
training_only (*evalml.pipelines.components.ComponentBase property*), 1355
TrainingValidationSplit (*class in evalml.preprocessing*), 1874
TrainingValidationSplit (*class in evalml.preprocessing.data_splitters*), 1864
TrainingValidationSplit (*class in evalml.preprocessing.data_splitters.training_validation_split*), 1858
transform() (*evalml.pipelines.binary_classification_pipeline.BinaryClassificationPipeline method*), 1560
transform() (*evalml.pipelines.BinaryClassificationPipeline method*), 1652
transform() (*evalml.pipelines.classification_pipeline.ClassificationPipeline method*), 1567
transform() (*evalml.pipelines.ClassificationPipeline method*), 1664
transform() (*evalml.pipelines.component_graph.ComponentGraph method*), 1573
transform() (*evalml.pipelines.ComponentGraph method*), 1670
transform() (*evalml.pipelines.components.DateTimeFeaturizer method*), 1359
transform() (*evalml.pipelines.components.DFSTransformer method*), 1369
transform() (*evalml.pipelines.components.DropColumns method*), 1371
transform() (*evalml.pipelines.components.DropNaNRowsTransformer method*), 1373
transform() (*evalml.pipelines.components.DropNullColumns method*), 1376
transform() (*evalml.pipelines.components.DropRowsTransformer method*), 1378
transform() (*evalml.pipelines.components.EmailFeaturizer method*), 1387
transform() (*evalml.pipelines.components.FeatureSelector method*), 1404
transform() (*evalml.pipelines.components.Imputer method*), 1406

transform() (*evalml.pipelines.components.LabelEncoder method*), 1412
transform() (*evalml.pipelines.components.LinearDiscriminantAnalysis method*), 1422
transform() (*evalml.pipelines.components.LogTransformer method*), 1431
transform() (*evalml.pipelines.components.LSA method*), 1434
transform() (*evalml.pipelines.components.NaturalLanguageFeaturizer method*), 1436
transform() (*evalml.pipelines.components.OneHotEncoder method*), 1440
transform() (*evalml.pipelines.components.OrdinalEncoder method*), 1443
transform() (*evalml.pipelines.components.Oversampler method*), 1445
transform() (*evalml.pipelines.components.PCA method*), 1448
transform() (*evalml.pipelines.components.PerColumnImputer method*), 1450
transform() (*evalml.pipelines.components.PolynomialDecomposer method*), 1455
transform() (*evalml.pipelines.components.ReplaceNullableTypes method*), 1467
transform() (*evalml.pipelines.components.RFClassifierRFESelector method*), 1470
transform() (*evalml.pipelines.components.RFClassifierSelectFromModel method*), 1473
transform() (*evalml.pipelines.components.RFRegressorRFESelector method*), 1476
transform() (*evalml.pipelines.components.RFRegressorSelectFromModel method*), 1479
transform() (*evalml.pipelines.components.SelectByType method*), 1482
transform() (*evalml.pipelines.components.SelectColumns method*), 1484
transform() (*evalml.pipelines.components.SimpleImputer method*), 1486
transform() (*evalml.pipelines.components.StandardScaler method*), 1499
transform() (*evalml.pipelines.components.STLDecomposer method*), 1504
transform() (*evalml.pipelines.components.TargetEncoder method*), 1514
transform() (*evalml.pipelines.components.TargetImputer method*), 1516
transform() (*evalml.pipelines.components.TimeSeriesFeaturizer method*), 1523
transform() (*evalml.pipelines.components.TimeSeriesImputer method*), 1525
transform() (*evalml.pipelines.components.TimeSeriesRegularizer method*), 1529
transform() (*evalml.pipelines.components.Transformer method*), 1531
transform() (*evalml.pipelines.components.transformers.column_selectors method*), 1203
transform() (*evalml.pipelines.components.transformers.column_selectors method*), 1205
transform() (*evalml.pipelines.components.transformers.column_selectors method*), 1208
transform() (*evalml.pipelines.components.transformers.column_selectors method*), 1210
transform() (*evalml.pipelines.components.transformers.DateTimeFeaturi method*), 1218
transform() (*evalml.pipelines.components.transformers.DFSTransformer method*), 1220
transform() (*evalml.pipelines.components.transformers.dimensionality_r method*), 959
transform() (*evalml.pipelines.components.transformers.dimensionality_r method*), 965
transform() (*evalml.pipelines.components.transformers.dimensionality_r method*), 968
transform() (*evalml.pipelines.components.transformers.dimensionality_r method*), 962
transform() (*evalml.pipelines.components.transformers.DropColumns method*), 1223
transform() (*evalml.pipelines.components.transformers.DropNaNRowsTr method*), 1225
transform() (*evalml.pipelines.components.transformers.DropNullColumn method*), 1228
transform() (*evalml.pipelines.components.transformers.DropRowsTransfo method*), 1230
transform() (*evalml.pipelines.components.transformers.EmailFeaturizer method*), 1232
transform() (*evalml.pipelines.components.transformers.encoders.label_e method*), 971
transform() (*evalml.pipelines.components.transformers.encoders.LabelE method*), 986
transform() (*evalml.pipelines.components.transformers.encoders.onehot_ method*), 975
transform() (*evalml.pipelines.components.transformers.encoders.OneHo method*), 990
transform() (*evalml.pipelines.components.transformers.encoders.ordinal method*), 979
transform() (*evalml.pipelines.components.transformers.encoders.Ordina method*), 993
transform() (*evalml.pipelines.components.transformers.encoders.target_ method*), 983
transform() (*evalml.pipelines.components.transformers.encoders.TargetE method*), 996
transform() (*evalml.pipelines.components.transformers.feature_selection method*), 999
transform() (*evalml.pipelines.components.transformers.feature_selection method*), 1019
transform() (*evalml.pipelines.components.transformers.feature_selection method*), 1002
transform() (*evalml.pipelines.components.transformers.feature_selection method*), 1005

transform() (*evalml.pipelines.components.transformers.feature_selection.RFCLassifierSelectFromModel method*), 1009
transform() (*evalml.pipelines.components.transformers.feature_selection.RFCLassifierSelectFromModel method*), 1012
transform() (*evalml.pipelines.components.transformers.feature_selection.RFRegressorSelectFromModel method*), 1016
transform() (*evalml.pipelines.components.transformers.feature_selection.RFRegressorSelectFromModel method*), 1022
transform() (*evalml.pipelines.components.transformers.feature_selection.RFRegressorSelectFromModel method*), 1025
transform() (*evalml.pipelines.components.transformers.feature_selection.RFRegressorSelectFromModel method*), 1028
transform() (*evalml.pipelines.components.transformers.feature_selection.RFRegressorSelectFromModel method*), 1032
transform() (*evalml.pipelines.components.transformers.FeatureSelection method*), 1235
transform() (*evalml.pipelines.components.transformers.Imputer method*), 1238
transform() (*evalml.pipelines.components.transformers.imputers.BaseImputer method*), 1054
transform() (*evalml.pipelines.components.transformers.imputers.BaseImputer method*), 1035
transform() (*evalml.pipelines.components.transformers.imputers.KNNImputer method*), 1038
transform() (*evalml.pipelines.components.transformers.imputers.KNNImputer method*), 1057
transform() (*evalml.pipelines.components.transformers.imputers.PerColumnImputer method*), 1041
transform() (*evalml.pipelines.components.transformers.imputers.PerColumnImputer method*), 1059
transform() (*evalml.pipelines.components.transformers.imputers.SimpleImputer method*), 1044
transform() (*evalml.pipelines.components.transformers.imputers.SimpleImputer method*), 1062
transform() (*evalml.pipelines.components.transformers.imputers.TimeSeriesImputer method*), 1047
transform() (*evalml.pipelines.components.transformers.imputers.TimeSeriesImputer method*), 1064
transform() (*evalml.pipelines.components.transformers.imputers.TimeSeriesImputer method*), 1051
transform() (*evalml.pipelines.components.transformers.imputers.TimeSeriesImputer method*), 1067
transform() (*evalml.pipelines.components.transformers.LabelEncoder method*), 1241
transform() (*evalml.pipelines.components.transformers.LinearDiscriminantAnalysis method*), 1243
transform() (*evalml.pipelines.components.transformers.LogTransformer method*), 1245
transform() (*evalml.pipelines.components.transformers.LSA method*), 1248
transform() (*evalml.pipelines.components.transformers.NaturalLanguageFeature method*), 1251
transform() (*evalml.pipelines.components.transformers.OneHotEncoder method*), 1254
transform() (*evalml.pipelines.components.transformers.recursive_elimination_selectors.RFCLassifierRecursiveFeatureElimination method*), 1257
transform() (*evalml.pipelines.components.transformers.feature_selection.RFCLassifierSelectFromModel method*), 1260
transform() (*evalml.pipelines.components.transformers.feature_selection.RFRegressorSelectFromModel method*), 1262
transform() (*evalml.pipelines.components.transformers.PerColumnImputer method*), 1265
transform() (*evalml.pipelines.components.transformers.PolynomialDecomposer method*), 1270
transform() (*evalml.pipelines.components.transformers.preprocessing.DateTimeFeaturizer method*), 1071
transform() (*evalml.pipelines.components.transformers.preprocessing.DelayedFeatureTransformer method*), 1131
transform() (*evalml.pipelines.components.transformers.preprocessing.DelayedFeatureTransformer method*), 1135
transform() (*evalml.pipelines.components.transformers.preprocessing.delayed_feature_extractor.DelayedFeatureExtractor method*), 1075
transform() (*evalml.pipelines.components.transformers.preprocessing.DropColumns method*), 1138
transform() (*evalml.pipelines.components.transformers.preprocessing.drop_rows_transformer.DropRowsTransformer method*), 1078
transform() (*evalml.pipelines.components.transformers.preprocessing.drop_rows_transformer.DropRowsTransformer method*), 1081
transform() (*evalml.pipelines.components.transformers.preprocessing.drop_rows_transformer.DropRowsTransformer method*), 1084
transform() (*evalml.pipelines.components.transformers.preprocessing.DropNullColumns method*), 1140
transform() (*evalml.pipelines.components.transformers.preprocessing.DropNullColumns method*), 1142
transform() (*evalml.pipelines.components.transformers.preprocessing.DropNullRows method*), 1145
transform() (*evalml.pipelines.components.transformers.preprocessing.EmptyDataCheckPreprocessor method*), 1147
transform() (*evalml.pipelines.components.transformers.preprocessing.feature_selection.FeatureSelectionTransformer method*), 1087
transform() (*evalml.pipelines.components.transformers.preprocessing.log_transformer.LogTransformer method*), 1090
transform() (*evalml.pipelines.components.transformers.preprocessing.LowVarianceDataCheck method*), 1150
transform() (*evalml.pipelines.components.transformers.preprocessing.LSA method*), 1152
transform() (*evalml.pipelines.components.transformers.preprocessing.lsa.LSATransformer method*), 1093
transform() (*evalml.pipelines.components.transformers.preprocessing.natural_language.NaturalLanguageFeaturizer method*), 1096
transform() (*evalml.pipelines.components.transformers.preprocessing.NaturalLanguageFeaturizer method*), 1155
transform() (*evalml.pipelines.components.transformers.preprocessing.polynomial_decomposer.PolynomialDecomposer method*), 1102
transform() (*evalml.pipelines.components.transformers.preprocessing.PolynomialDecomposer method*), 1160
transform() (*evalml.pipelines.components.transformers.preprocessing.replace_nan_transformer.ReplaceNanTransformer method*), 1105

transform() (evalml.pipelines.components.transformers.preprocessing.RemoveNullableTypes method), 1162
transform() (evalml.pipelines.components.transformers.preprocessing.TextTransformer method), 1111
transform() (evalml.pipelines.components.transformers.preprocessing.TimeSeriesFeaturizer method), 1167
transform() (evalml.pipelines.components.transformers.preprocessing.TimeSeriesImputer method), 1114
transform() (evalml.pipelines.components.transformers.preprocessing.TimeSeriesRegularizer method), 1170
transform() (evalml.pipelines.components.transformers.preprocessing.TimeSeriesRemoverer method), 1118
transform() (evalml.pipelines.components.transformers.preprocessing.TimeSeriesRemoverer method), 1122
transform() (evalml.pipelines.components.transformers.preprocessing.TimeSeriesSplitter method), 1173
transform() (evalml.pipelines.components.transformers.preprocessing.TimeSeriesSplitter method), 1177
transform() (evalml.pipelines.components.transformers.preprocessing.TrainingValidationSplit method), 1125
transform() (evalml.pipelines.components.transformers.preprocessing.UnstackTwoLevelIndex method), 1127
transform() (evalml.pipelines.components.transformers.preprocessing.UrlFeaturizer method), 1179
transform() (evalml.pipelines.components.transformers.ReplaceNullableTypes method), 1273
transform() (evalml.pipelines.components.transformers.RFClassifierSelectFromModel method), 1276
transform() (evalml.pipelines.components.transformers.RFClassifierSelectFromModel method), 1279
transform() (evalml.pipelines.components.transformers.RFRegressorSelectFromModel method), 1282
transform() (evalml.pipelines.components.transformers.RFRegressorSelectFromModel method), 1285
transform() (evalml.pipelines.components.transformers.samplers.base_sampler.BaseSampler method), 1182
transform() (evalml.pipelines.components.transformers.samplers.OneHotSampler method), 1192
transform() (evalml.pipelines.components.transformers.samplers.oversampler.Oversampler method), 1185
transform() (evalml.pipelines.components.transformers.samplers.Undersampler method), 1195
transform() (evalml.pipelines.components.transformers.samplers.undersampler.Undersampler method), 1189
transform() (evalml.pipelines.components.transformers.scalers.standard_scaler.StandardScaler method), 1197
transform() (evalml.pipelines.components.transformers.scalers.StandardScaler method), 1200
transform() (evalml.pipelines.components.transformers.SelectByType method), 1287
transform() (evalml.pipelines.components.transformers.SelectColumns method), 1289
transform() (evalml.pipelines.components.transformers.SimpleImputer method), 1292
transform() (evalml.pipelines.components.transformers.StandardScaler method), 1294
transform() (evalml.pipelines.components.transformers.STLDecomposer method), 1299
transform() (evalml.pipelines.components.transformers.TargetEncoder method), 1302
transform() (evalml.pipelines.components.transformers.TargetImputer method), 1305
transform() (evalml.pipelines.components.transformers.TextTransformer method), 1308
transform() (evalml.pipelines.components.transformers.TimeSeriesFeaturizer method), 1311
transform() (evalml.pipelines.components.transformers.TimeSeriesImputer method), 1314
transform() (evalml.pipelines.components.transformers.TimeSeriesRegularizer method), 1317
transform() (evalml.pipelines.components.transformers.transformer.Transformer method), 1213
transform() (evalml.pipelines.components.transformers.UnstackTwoLevelIndex method), 1320
transform() (evalml.pipelines.components.UrlFeaturizer method), 1323
transform() (evalml.pipelines.components.Undersampler method), 1534
transform() (evalml.pipelines.components.URLFeaturizer method), 1537
transform() (evalml.pipelines.DFSTransformer method), 1680
transform() (evalml.pipelines.DropNaNRowsTransformer method), 1683
transform() (evalml.pipelines.FeatureSelector method), 1706
transform() (evalml.pipelines.Imputer method), 1709
transform() (evalml.pipelines.multiclass_classification_pipeline.MulticlassClassificationPipeline method), 1732
transform() (evalml.pipelines.OneHotEncoder method), 1736
transform() (evalml.pipelines.OrdinalEncoder method), 1739
transform() (evalml.pipelines.PerColumnImputer method), 1741
transform() (evalml.pipelines.pipeline_base.PipelineBase method), 1587
transform() (evalml.pipelines.PipelineBase method), 1747
transform() (evalml.pipelines.regression_pipeline.RegressionPipeline method), 1595
transform() (evalml.pipelines.RegressionPipeline method), 1763
transform() (evalml.pipelines.RFClassifierSelectFromModel method), 1766
transform() (evalml.pipelines.RFRegressorSelectFromModel

*method*), 1769
transform() (*evalml.pipelines.SimpleImputer method*), 1771
transform() (*evalml.pipelines.StandardScaler method*), 1784
transform() (*evalml.pipelines.TargetEncoder method*), 1794
transform() (*evalml.pipelines.time_series_classification_pipelines.TimeSeriesBinaryClassificationPipeline method*), 1603
transform() (*evalml.pipelines.time_series_classification_pipelines.TimeSeriesMulticlassClassificationPipeline method*), 1611
transform() (*evalml.pipelines.time_series_classification_pipelines.TimeSeriesRegressionPipeline method*), 1618
transform() (*evalml.pipelines.time_series_pipeline_base.TimeSeriesPipelineBase method*), 1626
transform() (*evalml.pipelines.time_series_regression_pipeline.TimeSeriesRegressionPipeline method*), 1635
transform() (*evalml.pipelines.TimeSeriesBinaryClassificationPipeline method*), 1801
transform() (*evalml.pipelines.TimeSeriesClassificationPipeline method*), 1809
transform() (*evalml.pipelines.TimeSeriesFeaturizer method*), 1812
transform() (*evalml.pipelines.TimeSeriesImputer method*), 1815
transform() (*evalml.pipelines.TimeSeriesMulticlassClassificationPipeline method*), 1822
transform() (*evalml.pipelines.TimeSeriesRegressionPipeline method*), 1830
transform() (*evalml.pipelines.TimeSeriesRegularizer method*), 1834
transform() (*evalml.pipelines.Transformer method*), 1836
transform_all_but_final() (*evalml.pipelines.binary_classification_pipeline.BinaryClassificationPipeline method*), 1560
transform_all_but_final() (*evalml.pipelines.BinaryClassificationPipeline method*), 1652
transform_all_but_final() (*evalml.pipelines.classification_pipeline.ClassificationPipeline method*), 1568
transform_all_but_final() (*evalml.pipelines.ClassificationPipeline method*), 1665
transform_all_but_final() (*evalml.pipelines.component_graph.ComponentGraph method*), 1573
transform_all_but_final() (*evalml.pipelines.ComponentGraph method*), 1670
transform_all_but_final() (*evalml.pipelines.multiclass_classification_pipeline.MulticlassClassificationPipeline method*), 1581
transform_all_but_final() (*evalml.pipelines.MulticlassClassificationPipeline method*), 1733
transform_all_but_final() (*evalml.pipelines.pipeline_base.PipelineBase method*), 1587
transform_all_but_final() (*evalml.pipelines.PipelineBase method*), 1747
transform_all_but_final() (*evalml.pipelines.regression_pipeline.RegressionPipeline method*), 1595
transform_all_but_final() (*evalml.pipelines.RegressionPipeline method*), 1763
transform_all_but_final() (*evalml.pipelines.time_series_classification_pipelines.TimeSeriesBinaryClassificationPipeline method*), 1604
transform_all_but_final() (*evalml.pipelines.time_series_classification_pipelines.TimeSeriesMulticlassClassificationPipeline method*), 1611
transform_all_but_final() (*evalml.pipelines.time_series_classification_pipelines.TimeSeriesRegressionPipeline method*), 1619
transform_all_but_final() (*evalml.pipelines.time_series_pipeline_base.TimeSeriesPipelineBase method*), 1626
transform_all_but_final() (*evalml.pipelines.time_series_regression_pipeline.TimeSeriesRegressionPipeline method*), 1635
transform_all_but_final() (*evalml.pipelines.TimeSeriesBinaryClassificationPipeline method*), 1802
transform_all_but_final() (*evalml.pipelines.TimeSeriesClassificationPipeline method*), 1809
transform_all_but_final() (*evalml.pipelines.TimeSeriesMulticlassClassificationPipeline method*), 1822
transform_all_but_final() (*evalml.pipelines.TimeSeriesRegressionPipeline method*), 1831
Transformer (*class in evalml.pipelines*), 1834
Transformer (*class in evalml.pipelines.components*), 1529
Transformer (*class in evalml.pipelines.components.transformers*), 1315
Transformer (*class in evalml.pipelines.components.transformers.transformer*), 1211
tune_binary_threshold() (*in module evalml.automl*), 285
tune_binary_threshold() (*in module*

*evalml.automl.utils*), 272
Tuner (*class in evalml.tuners*), 1896
Tuner (*class in evalml.tuners.tuner*), 1890

## U

Undersampler (*class in evalml.pipelines.components*), 1532
Undersampler (*class in evalml.pipelines.components.transformers*), 1318
Undersampler (*class in evalml.pipelines.components.transformers.samplers*), 1192
Undersampler (*class in evalml.pipelines.components.transformers.samplers.undersampler*), 1186
uniqueness_score() (*evalml.data_checks.uniqueness_data_check.UniquenessDataCheck static method*), 338
uniqueness_score() (*evalml.data_checks.UniquenessDataCheck static method*), 386
UniquenessDataCheck (*class in evalml.data_checks*), 385
UniquenessDataCheck (*class in evalml.data_checks.uniqueness_data_check*), 338
update() (*evalml.automl.pipeline_search_plots.SearchIterationPlot method*), 267
update_parameters() (*evalml.pipelines.ARIMARegressor method*), 1645
update_parameters() (*evalml.pipelines.CatBoostClassifier method*), 1655
update_parameters() (*evalml.pipelines.CatBoostRegressor method*), 1658
update_parameters() (*evalml.pipelines.components.ARIMARegressor method*), 1340
update_parameters() (*evalml.pipelines.components.BaselineClassifier method*), 1343
update_parameters() (*evalml.pipelines.components.BaselineRegressor method*), 1346
update_parameters() (*evalml.pipelines.components.CatBoostClassifier method*), 1350
update_parameters() (*evalml.pipelines.components.CatBoostRegressor method*), 1353
update_parameters() (*evalml.pipelines.components.component_base.ComponentBase method*), 1326
update_parameters() (*evalml.pipelines.components.ComponentBase method*), 1355
update_parameters() (*evalml.pipelines.components.DateTimeFeaturizer method*), 1359
update_parameters() (*evalml.pipelines.components.DecisionTreeClassifier method*), 1362
update_parameters() (*evalml.pipelines.components.DecisionTreeRegressor method*), 1366
update_parameters() (*evalml.pipelines.components.DFSTransformer method*), 1369
update_parameters() (*evalml.pipelines.components.DropColumns method*), 1371
update_parameters() (*evalml.pipelines.components.DropNaNRowsTransformer method*), 1374
update_parameters() (*evalml.pipelines.components.DropNullColumns method*), 1376
update_parameters() (*evalml.pipelines.components.DropRowsTransformer method*), 1379
update_parameters() (*evalml.pipelines.components.ElasticNetClassifier method*), 1382
update_parameters() (*evalml.pipelines.components.ElasticNetRegressor method*), 1385
update_parameters() (*evalml.pipelines.components.EmailFeaturizer method*), 1387
update_parameters() (*evalml.pipelines.components.ensemble.stacked_ensemble_base.S method*), 622
update_parameters() (*evalml.pipelines.components.ensemble.stacked_ensemble_classif method*), 626
update_parameters() (*evalml.pipelines.components.ensemble.stacked_ensemble_regres method*), 630
update_parameters() (*evalml.pipelines.components.ensemble.StackedEnsembleBase method*), 634
update_parameters() (*evalml.pipelines.components.ensemble.StackedEnsembleClassifie method*), 638
update_parameters() (*evalml.pipelines.components.ensemble.StackedEnsembleRegress method*), 642

update_parameters() (evalml.pipelines.components.Estimator method), 1391
update_parameters() (evalml.pipelines.components.estimators.ARIMARegressor method), 862
update_parameters() (evalml.pipelines.components.estimators.BaselineClassifier method), 865
update_parameters() (evalml.pipelines.components.estimators.BaselineRegressor method), 868
update_parameters() (evalml.pipelines.components.estimators.CatBoostClassifier method), 872
update_parameters() (evalml.pipelines.components.estimators.CatBoostRegressor method), 875
update_parameters() (evalml.pipelines.components.estimators.classifiers.baseline_classifier.BaselineClassifier method), 646
update_parameters() (evalml.pipelines.components.estimators.classifiers.BaselineClassifier method), 699
update_parameters() (evalml.pipelines.components.estimators.classifiers.catboost_classifier.CatBoostClassifier method), 650
update_parameters() (evalml.pipelines.components.estimators.classifiers.CatBoostClassifier method), 702
update_parameters() (evalml.pipelines.components.estimators.classifiers.decision_tree_classifier.DecisionTreeClassifier method), 654
update_parameters() (evalml.pipelines.components.estimators.classifiers.DecisionTreeClassifier method), 706
update_parameters() (evalml.pipelines.components.estimators.classifiers.elasticnet_classifier.ElasticNetClassifier method), 658
update_parameters() (evalml.pipelines.components.estimators.classifiers.ElasticNetClassifier method), 709
update_parameters() (evalml.pipelines.components.estimators.classifiers.et_classifier_sklearn.ExtraTreesClassifier method), 662
update_parameters() (evalml.pipelines.components.estimators.classifiers.ExtraTreesClassifier method), 713
update_parameters() (evalml.pipelines.components.estimators.classifiers.kneighbors_classifier.KNeighborsClassifier method), 666
update_parameters() (evalml.pipelines.components.estimators.classifiers.KNeighborsClassifier method), 717
update_parameters() (evalml.pipelines.components.estimators.classifiers.lightgbm_clas method), 670
update_parameters() (evalml.pipelines.components.estimators.classifiers.LightGBMCla method), 720
update_parameters() (evalml.pipelines.components.estimators.classifiers.logistic_regre method), 674
update_parameters() (evalml.pipelines.components.estimators.classifiers.LogisticRegre method), 724
update_parameters() (evalml.pipelines.components.estimators.classifiers.RandomFores method), 727
update_parameters() (evalml.pipelines.components.estimators.classifiers.rf_classifier.R method), 678
update_parameters() (evalml.pipelines.components.estimators.classifiers.svm_classifier method), 681
update_parameters() (evalml.pipelines.components.estimators.classifiers.SVMClassifie method), 730
update_parameters() (evalml.pipelines.components.estimators.classifiers.vowpal_wabb method), 685
update_parameters() (evalml.pipelines.components.estimators.classifiers.vowpal_wabb method), 688
update_parameters() (evalml.pipelines.components.estimators.classifiers.vowpal_wabb method), 692
update_parameters() (evalml.pipelines.components.estimators.classifiers.VowpalWabbi method), 734
update_parameters() (evalml.pipelines.components.estimators.classifiers.VowpalWabbi method), 737
update_parameters() (evalml.pipelines.components.estimators.classifiers.xgboost_class method), 695
update_parameters() (evalml.pipelines.components.estimators.classifiers.XGBoostClas method), 740
update_parameters() (evalml.pipelines.components.estimators.DecisionTreeClassifier method), 879
update_parameters() (evalml.pipelines.components.estimators.DecisionTreeRegressor method), 883
update_parameters() (evalml.pipelines.components.estimators.ElasticNetClassifier method), 886

update_parameters() (evalml.pipelines.components.estimators.ElasticNetRegressor method), 889
update_parameters() (evalml.pipelines.components.estimators.Estimator method), 893
update_parameters() (evalml.pipelines.components.estimators.estimator.Estimator method), 857
update_parameters() (evalml.pipelines.components.estimators.ExponentialSmoothingRegressor method), 896
update_parameters() (evalml.pipelines.components.estimators.ExtraTreesClassifier method), 900
update_parameters() (evalml.pipelines.components.estimators.ExtraTreesRegressor method), 904
update_parameters() (evalml.pipelines.components.estimators.KNeighborsClassifier method), 907
update_parameters() (evalml.pipelines.components.estimators.LightGBMClassifier method), 911
update_parameters() (evalml.pipelines.components.estimators.LightGBMRegressor method), 915
update_parameters() (evalml.pipelines.components.estimators.LinearRegressor method), 918
update_parameters() (evalml.pipelines.components.estimators.LogisticRegressionClassifier method), 921
update_parameters() (evalml.pipelines.components.estimators.ProphetRegressor method), 925
update_parameters() (evalml.pipelines.components.estimators.RandomForestClassifier method), 928
update_parameters() (evalml.pipelines.components.estimators.RandomForestRegressor method), 931
update_parameters() (evalml.pipelines.components.estimators.regressors.arima_regressor.ARIMARegressor method), 746
update_parameters() (evalml.pipelines.components.estimators.regressors.ARIMARegressor method), 805
update_parameters() (evalml.pipelines.components.estimators.regressors.baseline_regressor.BaselineRegressor method), 749
update_parameters() (evalml.pipelines.components.estimators.regressors.BaselineRegressor method), 808
update_parameters() (evalml.pipelines.components.estimators.regressors.catboost_regr method), 753
update_parameters() (evalml.pipelines.components.estimators.regressors.CatBoostRegr method), 812
update_parameters() (evalml.pipelines.components.estimators.regressors.decision_tree method), 757
update_parameters() (evalml.pipelines.components.estimators.regressors.DecisionTree method), 815
update_parameters() (evalml.pipelines.components.estimators.regressors.elasticnet_reg method), 761
update_parameters() (evalml.pipelines.components.estimators.regressors.ElasticNetReg method), 819
update_parameters() (evalml.pipelines.components.estimators.regressors.et_regressor.E method), 765
update_parameters() (evalml.pipelines.components.estimators.regressors.exponential_s method), 769
update_parameters() (evalml.pipelines.components.estimators.regressors.ExponentialS method), 822
update_parameters() (evalml.pipelines.components.estimators.regressors.ExtraTreesRe method), 826
update_parameters() (evalml.pipelines.components.estimators.regressors.lightgbm_reg method), 773
update_parameters() (evalml.pipelines.components.estimators.regressors.LightGBMRe method), 830
update_parameters() (evalml.pipelines.components.estimators.regressors.linear_regres method), 777
update_parameters() (evalml.pipelines.components.estimators.regressors.LinearRegres method), 833
update_parameters() (evalml.pipelines.components.estimators.regressors.prophet_regr method), 782
update_parameters() (evalml.pipelines.components.estimators.regressors.ProphetRegre method), 837
update_parameters() (evalml.pipelines.components.estimators.regressors.RandomFores method), 840
update_parameters() (evalml.pipelines.components.estimators.regressors.rf_regressor.R method), 786

update_parameters() (evalml.pipelines.components.estimators.regressors.svm_regressor.SVMRegressor method), 789
update_parameters() (evalml.pipelines.components.estimators.regressors.SVMRegressor method), 843
update_parameters() (evalml.pipelines.components.estimators.regressors.time_series_baseline_estimator.TimeSeriesBaselineEstimator method), 793
update_parameters() (evalml.pipelines.components.estimators.regressors.TimeSeriesBaselineEstimator method), 846
update_parameters() (evalml.pipelines.components.estimators.regressors.vowpal_wabbit_regressor.VowpalWabbitBaseRegressor method), 796
update_parameters() (evalml.pipelines.components.estimators.regressors.VowpalWabbitBaseRegressor method), 850
update_parameters() (evalml.pipelines.components.estimators.regressors.xgboost_regressor.XGBoostRegressor method), 800
update_parameters() (evalml.pipelines.components.estimators.regressors.XGBoostRegressor method), 853
update_parameters() (evalml.pipelines.components.estimators.SVMClassifier method), 934
update_parameters() (evalml.pipelines.components.estimators.SVMRegressor method), 937
update_parameters() (evalml.pipelines.components.estimators.TimeSeriesBaselineEstimator method), 940
update_parameters() (evalml.pipelines.components.estimators.VowpalWabbitBinaryClassifier method), 944
update_parameters() (evalml.pipelines.components.estimators.VowpalWabbitMulticlassClassifier method), 947
update_parameters() (evalml.pipelines.components.estimators.VowpalWabbitRegressor method), 950
update_parameters() (evalml.pipelines.components.estimators.XGBoostClassifier method), 953
update_parameters() (evalml.pipelines.components.estimators.XGBoostRegressor method), 956
update_parameters() (evalml.pipelines.components.ExponentialSmoothingRegressor method), 1394
update_parameters() (evalml.pipelines.components.ExtraTreesClassifier method), 1398
update_parameters() (evalml.pipelines.components.ExtraTreesRegressor method), 1401
update_parameters() (evalml.pipelines.components.FeatureSelector method), 1404
update_parameters() (evalml.pipelines.components.TimeSeriesBaselineEstimator method), 1407
update_parameters() (evalml.pipelines.components.KNeighborsClassifier method), 1410
update_parameters() (evalml.pipelines.components.VowpalWabbitLabelEncoder method), 1413
update_parameters() (evalml.pipelines.components.LightGBMClassifier method), 1416
update_parameters() (evalml.pipelines.components.LightGBMRegressor method), 1420
update_parameters() (evalml.pipelines.components.LinearDiscriminantAnalysis method), 1422
update_parameters() (evalml.pipelines.components.LinearRegressor method), 1425
update_parameters() (evalml.pipelines.components.LogisticRegressionClassifier method), 1429
update_parameters() (evalml.pipelines.components.LogTransformer method), 1431
update_parameters() (evalml.pipelines.components.LSA method), 1434
update_parameters() (evalml.pipelines.components.NaturalLanguageFeaturizer method), 1436
update_parameters() (evalml.pipelines.components.OneHotEncoder method), 1440
update_parameters() (evalml.pipelines.components.OrdinalEncoder method), 1443
update_parameters() (evalml.pipelines.components.Oversampler method), 1446
update_parameters() (evalml.pipelines.components.PCA method), 1448
update_parameters() (evalml.pipelines.components.PerColumnImputer method), 1451

update_parameters() (*evalml.pipelines.components.PolynomialDecomposer method*), 1456
update_parameters() (*evalml.pipelines.components.ProphetRegressor method*), 1459
update_parameters() (*evalml.pipelines.components.RandomForestClassifier method*), 1462
update_parameters() (*evalml.pipelines.components.RandomForestRegressor method*), 1465
update_parameters() (*evalml.pipelines.components.ReplaceNullableTypes method*), 1467
update_parameters() (*evalml.pipelines.components.RFClassifierRFESelector method*), 1470
update_parameters() (*evalml.pipelines.components.RFClassifierSelectFromModel method*), 1473
update_parameters() (*evalml.pipelines.components.RFRegressorRFESelector method*), 1476
update_parameters() (*evalml.pipelines.components.RFRegressorSelectFromModel method*), 1479
update_parameters() (*evalml.pipelines.components.SelectByType method*), 1482
update_parameters() (*evalml.pipelines.components.SelectColumns method*), 1484
update_parameters() (*evalml.pipelines.components.SimpleImputer method*), 1487
update_parameters() (*evalml.pipelines.components.StackedEnsembleBase method*), 1490
update_parameters() (*evalml.pipelines.components.StackedEnsembleClassifier method*), 1494
update_parameters() (*evalml.pipelines.components.StackedEnsembleRegressor method*), 1497
update_parameters() (*evalml.pipelines.components.StandardScaler method*), 1500
update_parameters() (*evalml.pipelines.components.STLDecomposer method*), 1505
update_parameters() (*evalml.pipelines.components.SVMClassifier method*), 1508
update_parameters() (*evalml.pipelines.components.SVMRegressor method*), 1511
update_parameters() (*evalml.pipelines.components.TargetEncoder method*), 1514
update_parameters() (*evalml.pipelines.components.TargetImputer method*), 1517
update_parameters() (*evalml.pipelines.components.TimeSeriesBaselineEstimator method*), 1520
update_parameters() (*evalml.pipelines.components.TimeSeriesFeaturizer method*), 1523
update_parameters() (*evalml.pipelines.components.TimeSeriesImputer method*), 1526
update_parameters() (*evalml.pipelines.components.TimeSeriesRegularizer method*), 1529
update_parameters() (*evalml.pipelines.components.Transformer method*), 1531
update_parameters() (*evalml.pipelines.components.transformers.column_selectors.Col method*), 1203
update_parameters() (*evalml.pipelines.components.transformers.column_selectors.Dro method*), 1206
update_parameters() (*evalml.pipelines.components.transformers.column_selectors.Sele method*), 1208
update_parameters() (*evalml.pipelines.components.transformers.column_selectors.Sele method*), 1210
update_parameters() (*evalml.pipelines.components.transformers.DateTimeFeaturizer method*), 1218
update_parameters() (*evalml.pipelines.components.transformers.DFSTransformer method*), 1221
update_parameters() (*evalml.pipelines.components.transformers.dimensionality_reduc method*), 959
update_parameters() (*evalml.pipelines.components.transformers.dimensionality_reduc method*), 965
update_parameters() (*evalml.pipelines.components.transformers.dimensionality_reduc method*), 968
update_parameters() (*evalml.pipelines.components.transformers.dimensionality_reduc method*), 962

update_parameters() (evalml.pipelines.components.transformers.DropColumns method), 1223
update_parameters() (evalml.pipelines.components.transformers.DropNaNRowsTransformer method), 1225
update_parameters() (evalml.pipelines.components.transformers.DropNullColumns method), 1228
update_parameters() (evalml.pipelines.components.transformers.DropRowsTransformer method), 1230
update_parameters() (evalml.pipelines.components.transformers.EmailFeaturizer method), 1233
update_parameters() (evalml.pipelines.components.transformers.encoders.label_encoder.LabelEncoder method), 971
update_parameters() (evalml.pipelines.components.transformers.encoders.LabelEncoder method), 986
update_parameters() (evalml.pipelines.components.transformers.encoders.onehot_encoder.OneHotEncoder method), 975
update_parameters() (evalml.pipelines.components.transformers.encoders.OneHotEncoder method), 990
update_parameters() (evalml.pipelines.components.transformers.encoders.ordinal_encoder.OrdinalEncoder method), 979
update_parameters() (evalml.pipelines.components.transformers.encoders.OrdinalEncoder method), 993
update_parameters() (evalml.pipelines.components.transformers.encoders.target_encoder.TargetEncoder method), 983
update_parameters() (evalml.pipelines.components.transformers.encoders.TargetEncoder method), 996
update_parameters() (evalml.pipelines.components.transformers.feature_selection.feature_selector.FeatureSelector method), 999
update_parameters() (evalml.pipelines.components.transformers.feature_selection.feature_selectors method), 1019
update_parameters() (evalml.pipelines.components.transformers.feature_selection.feature_elimination.transformer.RecursiveFeatureEliminator method), 1003
update_parameters() (evalml.pipelines.components.transformers.feature_selection.feature_elimination.transformer.RFClassificationFeatureEliminator method), 1006
update_parameters() (evalml.pipelines.components.transformers.feature_selection.feature_elimination.transformer.RFRegressionFeatureEliminator method), 1009
update_parameters() (evalml.pipelines.components.transformers.feature_selection.rf_c method), 1012
update_parameters() (evalml.pipelines.components.transformers.feature_selection.rf_r method), 1016
update_parameters() (evalml.pipelines.components.transformers.feature_selection.RFC method), 1022
update_parameters() (evalml.pipelines.components.transformers.feature_selection.RFC method), 1025
update_parameters() (evalml.pipelines.components.transformers.feature_selection.RFR method), 1029
update_parameters() (evalml.pipelines.components.transformers.feature_selection.RFR method), 1032
update_parameters() (evalml.pipelines.components.transformers.FeatureSelector method), 1236
update_parameters() (evalml.pipelines.components.transformers.Imputer method), 1238
update_parameters() (evalml.pipelines.components.transformers.imputers.Imputer method), 1055
update_parameters() (evalml.pipelines.components.transformers.imputers.imputer.Impu method), 1035
update_parameters() (evalml.pipelines.components.transformers.imputers.knn_imputer method), 1038
update_parameters() (evalml.pipelines.components.transformers.imputers.KNNImputer method), 1057
update_parameters() (evalml.pipelines.components.transformers.imputers.per_column_ method), 1041
update_parameters() (evalml.pipelines.components.transformers.imputers.PerColumnI method), 1059
update_parameters() (evalml.pipelines.components.transformers.imputers.simple_impu method), 1044
update_parameters() (evalml.pipelines.components.transformers.imputers.SimpleImput method), 1062
update_parameters() (evalml.pipelines.components.transformers.imputer method), 1047
update_parameters() (evalml.pipelines.components.transformers.imputer method), 1065

update_parameters() (evalml.pipelines.components.transformers.imputers.time_series_imputer.TimeSeriesImputer method), 1051
update_parameters() (evalml.pipelines.components.transformers.imputers.TimeSeriesImputer method), 1067
update_parameters() (evalml.pipelines.components.transformers.LabelEncoder method), 1241
update_parameters() (evalml.pipelines.components.transformers.LinearDiscriminantAnalysis method), 1243
update_parameters() (evalml.pipelines.components.transformers.LogTransformer method), 1246
update_parameters() (evalml.pipelines.components.transformers.LSA method), 1248
update_parameters() (evalml.pipelines.components.transformers.NaturalLanguageFeaturizer method), 1251
update_parameters() (evalml.pipelines.components.transformers.OneHotEncoder method), 1254
update_parameters() (evalml.pipelines.components.transformers.OrdinalEncoder method), 1257
update_parameters() (evalml.pipelines.components.transformers.Oversampler method), 1260
update_parameters() (evalml.pipelines.components.transformers.PCA method), 1263
update_parameters() (evalml.pipelines.components.transformers.PerColumnImputer method), 1265
update_parameters() (evalml.pipelines.components.transformers.PolynomialDecomposer method), 1271
update_parameters() (evalml.pipelines.components.transformers.preprocessing.datetime_featurizer.DateTimeFeaturizer method), 1071
update_parameters() (evalml.pipelines.components.transformers.preprocessing.DateTimeFeaturizer method), 1131
update_parameters() (evalml.pipelines.components.transformers.preprocessing.DelayedFeatureTransformer method), 1135
update_parameters() (evalml.pipelines.components.transformers.preprocessing.drop_columns_transformer.DropColumns method), 1076
update_parameters() (evalml.pipelines.components.transformers.preprocessing.DROPCOLUMNS method), 1138
update_parameters() (evalml.pipelines.components.transformers.preprocessing.drop_n method), 1078
update_parameters() (evalml.pipelines.components.transformers.preprocessing.drop_n method), 1081
update_parameters() (evalml.pipelines.components.transformers.preprocessing.drop_r method), 1084
update_parameters() (evalml.pipelines.components.transformers.preprocessing.DropNa method), 1140
update_parameters() (evalml.pipelines.components.transformers.preprocessing.DropNu method), 1143
update_parameters() (evalml.pipelines.components.transformers.preprocessing.DropRo method), 1145
update_parameters() (evalml.pipelines.components.transformers.preprocessing.EmailF method), 1148
update_parameters() (evalml.pipelines.components.transformers.preprocessing.feature method), 1088
update_parameters() (evalml.pipelines.components.transformers.preprocessing.log_tra method), 1090
update_parameters() (evalml.pipelines.components.transformers.preprocessing.LogTra method), 1150
update_parameters() (evalml.pipelines.components.transformers.preprocessing.LSA method), 1152
update_parameters() (evalml.pipelines.components.transformers.preprocessing.lsa.LSA method), 1093
update_parameters() (evalml.pipelines.components.transformers.preprocessing.natural method), 1096
update_parameters() (evalml.pipelines.components.transformers.preprocessing.NaturalLanguageFeaturizer method), 1155
update_parameters() (evalml.pipelines.components.transformers.preprocessing.polynomial_decomposer.PolynomialDecomposer method), 1103
update_parameters() (evalml.pipelines.components.transformers.preprocessing.PolynomialDecomposer method), 1160
update_parameters() (evalml.pipelines.components.transformers.preprocessing.replace_nan_operator.ReplaceNanOperator method), 1105
update_parameters() (evalml.pipelines.components.transformers.preprocessing.ReplaceNanOperator method), 1162

update_parameters() (evalml.pipelines.components.transformers.preprocessing.stl_decomposer.STLDecomposer method), 1112
update_parameters() (evalml.pipelines.components.transformers.preprocessing.STLDecomposition method), 1168
update_parameters() (evalml.pipelines.components.transformers.preprocessing.text_transformer.TextTransformer method), 1115
update_parameters() (evalml.pipelines.components.transformers.preprocessing.TextTransformer method), 1170
update_parameters() (evalml.pipelines.components.transformers.preprocessing.time_series_featurizer.TimeSeriesFeaturizer method), 1118
update_parameters() (evalml.pipelines.components.transformers.preprocessing.time_series_regularizer.TimeSeriesRegularizer method), 1122
update_parameters() (evalml.pipelines.components.transformers.preprocessing.TimeSeriesFeaturizer method), 1174
update_parameters() (evalml.pipelines.components.transformers.preprocessing.TimeSeriesRegularizer method), 1177
update_parameters() (evalml.pipelines.components.transformers.preprocessing.transform_primitive_components.EmailFeaturizer method), 1125
update_parameters() (evalml.pipelines.components.transformers.preprocessing.transform_primitive_components.URLFeaturizer method), 1127
update_parameters() (evalml.pipelines.components.transformers.preprocessing.URLFeaturizer method), 1179
update_parameters() (evalml.pipelines.components.transformers.ReplaceNullableTypes method), 1273
update_parameters() (evalml.pipelines.components.transformers.RFClassifierRFE.RFClassifierRFE method), 1276
update_parameters() (evalml.pipelines.components.transformers.RFClassifierSelectFromModel.RFClassifierSelectFromModel method), 1279
update_parameters() (evalml.pipelines.components.transformers.RFRegressorRFE.RFRegressorRFE method), 1282
update_parameters() (evalml.pipelines.components.transformers.RFRegressorSelectFromModel.RFRegressorSelectFromModel method), 1285
update_parameters() (evalml.pipelines.components.transformers.samplers.base_sampler.BaseSampler method), 1182
update_parameters() (evalml.pipelines.components.transformers.samplers.Oversampler method), 1192
update_parameters() (evalml.pipelines.components.transformers.samplers.oversampler.Oversampler method), 1185
update_parameters() (evalml.pipelines.components.transformers.samplers.Undersampler method), 1195
update_parameters() (evalml.pipelines.components.transformers.samplers.undersampler.Undersampler method), 1189
update_parameters() (evalml.pipelines.components.transformers.scalers.standard_scaler.StandardScaler method), 1198
update_parameters() (evalml.pipelines.components.transformers.scalers.StandardScaler method), 1200
update_parameters() (evalml.pipelines.components.transformers.SelectByType method), 1287
update_parameters() (evalml.pipelines.components.transformers.SelectColumns method), 1289
update_parameters() (evalml.pipelines.components.transformers.SimpleImputer method), 1292
update_parameters() (evalml.pipelines.components.transformers.StandardScaler method), 1294
update_parameters() (evalml.pipelines.components.transformers.STLDecomposer method), 1300
update_parameters() (evalml.pipelines.components.transformers.TargetEncoder method), 1303
update_parameters() (evalml.pipelines.components.transformers.TargetImputer method), 1305
update_parameters() (evalml.pipelines.components.transformers.TimeSeriesFeaturizer method), 1308
update_parameters() (evalml.pipelines.components.transformers.TimeSeriesImputer method), 1311
update_parameters() (evalml.pipelines.components.transformers.TimeSeriesRegularizer method), 1314
update_parameters() (evalml.pipelines.components.transformers.Transformer method), 1317
update_parameters() (evalml.pipelines.components.transformers.transformer.Transformer method), 1213
update_parameters() (evalml.pipelines.components.transformers.Undersampler method), 1321

update_parameters() (*evalml.pipelines.components.transformers.URLFeaturizer method*), 1323
update_parameters() (*evalml.pipelines.components.Undersampler method*), 1534
update_parameters() (*evalml.pipelines.components.URLFeaturizer method*), 1537
update_parameters() (*evalml.pipelines.components.VowpalWabbitBinaryClassifier method*), 1540
update_parameters() (*evalml.pipelines.components.VowpalWabbitMulticlassClassifier method*), 1543
update_parameters() (*evalml.pipelines.components.VowpalWabbitRegressor method*), 1546
update_parameters() (*evalml.pipelines.components.XGBoostClassifier method*), 1549
update_parameters() (*evalml.pipelines.components.XGBoostRegressor method*), 1552
update_parameters() (*evalml.pipelines.DecisionTreeClassifier method*), 1674
update_parameters() (*evalml.pipelines.DecisionTreeRegressor method*), 1678
update_parameters() (*evalml.pipelines.DFSTransformer method*), 1681
update_parameters() (*evalml.pipelines.DropNaNRowsTransformer method*), 1683
update_parameters() (*evalml.pipelines.ElasticNetClassifier method*), 1687
update_parameters() (*evalml.pipelines.ElasticNetRegressor method*), 1690
update_parameters() (*evalml.pipelines.Estimator method*), 1693
update_parameters() (*evalml.pipelines.ExponentialSmoothingRegressor method*), 1696
update_parameters() (*evalml.pipelines.ExtraTreesClassifier method*), 1700
update_parameters() (*evalml.pipelines.ExtraTreesRegressor method*), 1704
update_parameters() (*evalml.pipelines.FeatureSelector method*), 1707
update_parameters() (*evalml.pipelines.Imputer method*), 1709
update_parameters() (*evalml.pipelines.KNeighborsClassifier method*), 1713
update_parameters() (*evalml.pipelines.LightGBMClassifier method*), 1716
update_parameters() (*evalml.pipelines.LightGBMRegressor method*), 1720
update_parameters() (*evalml.pipelines.LinearRegressor method*), 1723
update_parameters() (*evalml.pipelines.LogisticRegressionClassifier method*), 1726
update_parameters() (*evalml.pipelines.OneHotEncoder method*), 1736
update_parameters() (*evalml.pipelines.OrdinalEncoder method*), 1739
update_parameters() (*evalml.pipelines.PerColumnImputer method*), 1742
update_parameters() (*evalml.pipelines.ProphetRegressor method*), 1751
update_parameters() (*evalml.pipelines.RandomForestClassifier method*), 1754
update_parameters() (*evalml.pipelines.RandomForestRegressor method*), 1757
update_parameters() (*evalml.pipelines.RFClassifierSelectFromModel method*), 1766
update_parameters() (*evalml.pipelines.RFRegressorSelectFromModel method*), 1769
update_parameters() (*evalml.pipelines.SimpleImputer method*), 1772
update_parameters() (*evalml.pipelines.StackedEnsembleBase method*), 1775
update_parameters() (*evalml.pipelines.StackedEnsembleClassifier method*), 1779
update_parameters() (*evalml.pipelines.StackedEnsembleRegressor

*method*), 1782
update_parameters() (*evalml.pipelines.StandardScaler method*), 1785
update_parameters() (*evalml.pipelines.SVMClassifier method*), 1788
update_parameters() (*evalml.pipelines.SVMRegressor method*), 1791
update_parameters() (*evalml.pipelines.TargetEncoder method*), 1794
update_parameters() (*evalml.pipelines.TimeSeriesFeaturizer method*), 1812
update_parameters() (*evalml.pipelines.TimeSeriesImputer method*), 1815
update_parameters() (*evalml.pipelines.TimeSeriesRegularizer method*), 1834
update_parameters() (*evalml.pipelines.Transformer method*), 1837
update_parameters() (*evalml.pipelines.VowpalWabbitBinaryClassifier method*), 1840
update_parameters() (*evalml.pipelines.VowpalWabbitMulticlassClassifier method*), 1843
update_parameters() (*evalml.pipelines.VowpalWabbitRegressor method*), 1846
update_parameters() (*evalml.pipelines.XGBoostClassifier method*), 1849
update_parameters() (*evalml.pipelines.XGBoostRegressor method*), 1852
URLFeaturizer (*class in evalml.pipelines.components*), 1535
URLFeaturizer (*class in evalml.pipelines.components.transformers*), 1321
URLFeaturizer (*class in evalml.pipelines.components.transformers.preprocessing*), 1177
URLFeaturizer (*class in evalml.pipelines.components.transformers.preprocessing.text_featurizers*), 1125


## V


validate() (*evalml.data_checks.class_imbalance_data_check.ClassImbalanceDataCheck method*), 287
validate() (*evalml.data_checks.ClassImbalanceDataCheck method*), 343
validate() (*evalml.data_checks.data_check.DataCheck method*), 289
validate() (*evalml.data_checks.data_checks.DataChecks method*), 299
validate() (*evalml.data_checks.DataCheck method*), 345
validate() (*evalml.data_checks.DataChecks method*), 351
validate() (*evalml.data_checks.datetime_format_data_check.DateTimeFo method*), 300
validate() (*evalml.data_checks.DateTimeFormatDataCheck method*), 352
validate() (*evalml.data_checks.default_data_checks.DefaultDataChecks method*), 308
validate() (*evalml.data_checks.DefaultDataChecks method*), 360
validate() (*evalml.data_checks.id_columns_data_check.IDColumnsData method*), 309
validate() (*evalml.data_checks.IDColumnsDataCheck method*), 361
validate() (*evalml.data_checks.invalid_target_data_check.InvalidTarget method*), 313
validate() (*evalml.data_checks.InvalidTargetDataCheck method*), 365
validate() (*evalml.data_checks.multicollinearity_data_check.Multicollin method*), 317
validate() (*evalml.data_checks.MulticollinearityDataCheck method*), 368
validate() (*evalml.data_checks.no_variance_data_check.NoVarianceDat method*), 318
validate() (*evalml.data_checks.NoVarianceDataCheck method*), 369
validate() (*evalml.data_checks.null_data_check.NullDataCheck method*), 322
validate() (*evalml.data_checks.NullDataCheck method*), 372
validate() (*evalml.data_checks.outliers_data_check.OutliersDataCheck method*), 326
validate() (*evalml.data_checks.OutliersDataCheck method*), 376
validate() (*evalml.data_checks.sparsity_data_check.SparsityDataCheck method*), 329
validate() (*evalml.data_checks.SparsityDataCheck method*), 378
validate() (*evalml.data_checks.target_distribution_data_check.TargetDis method*), 330
validate() (*evalml.data_checks.target_leakage_data_check.TargetLeakag method*), 332
validate() (*evalml.data_checks.TargetDistributionDataCheck method*), 379
validate() (*evalml.data_checks.TargetLeakageDataCheck method*), 381

validate() (*evalml.data_checks.TimeSeriesParametersDataCheck method*), 383
validate() (*evalml.data_checks.TimeSeriesSplittingDataCheck method*), 384
validate() (*evalml.data_checks.ts_parameters_data_check.TimeSeriesParametersDataCheck method*), 335
validate() (*evalml.data_checks.ts_splitting_data_check.TimeSeriesSplittingDataCheck method*), 336
validate() (*evalml.data_checks.uniqueness_data_check.UniquenessDataCheck method*), 339
validate() (*evalml.data_checks.UniquenessDataCheck method*), 386
validate_holdout_datasets() (*in module evalml.utils.gen_utils*), 1905
validate_inputs() (*evalml.objectives.AccuracyBinary method*), 536
validate_inputs() (*evalml.objectives.AccuracyMulticlass method*), 538
validate_inputs() (*evalml.objectives.AUC method*), 540
validate_inputs() (*evalml.objectives.AUCMacro method*), 542
validate_inputs() (*evalml.objectives.AUCMicro method*), 543
validate_inputs() (*evalml.objectives.AUCWeighted method*), 545
validate_inputs() (*evalml.objectives.BalancedAccuracyBinary method*), 547
validate_inputs() (*evalml.objectives.BalancedAccuracyMulticlass method*), 549
validate_inputs() (*evalml.objectives.binary_classification_objective.BinaryClassificationObjective method*), 443
validate_inputs() (*evalml.objectives.BinaryClassificationObjective method*), 552
validate_inputs() (*evalml.objectives.cost_benefit_matrix.CostBenefitMatrix method*), 446
validate_inputs() (*evalml.objectives.CostBenefitMatrix method*), 554
validate_inputs() (*evalml.objectives.ExpVariance method*), 556
validate_inputs() (*evalml.objectives.F1 method*), 558
validate_inputs() (*evalml.objectives.F1Macro method*), 560
validate_inputs() (*evalml.objectives.F1Micro method*), 561
validate_inputs() (*evalml.objectives.F1Weighted method*), 563
validate_inputs() (*evalml.objectives.fraud_cost.FraudCost method*), 449
validate_inputs() (*evalml.objectives.FraudCost method*), 566
validate_inputs() (*evalml.objectives.Gini method*), 569
validate_inputs() (*evalml.objectives.lead_scoring.LeadScoring method*), 451
validate_inputs() (*evalml.objectives.LeadScoring method*), 572
validate_inputs() (*evalml.objectives.LogLossBinary method*), 574
validate_inputs() (*evalml.objectives.LogLossMulticlass method*), 576
validate_inputs() (*evalml.objectives.MAE method*), 577
validate_inputs() (*evalml.objectives.MAPE method*), 579
validate_inputs() (*evalml.objectives.MaxError method*), 581
validate_inputs() (*evalml.objectives.MCCBinary method*), 583
validate_inputs() (*evalml.objectives.MCCMulticlass method*), 585
validate_inputs() (*evalml.objectives.MeanSquaredLogError method*), 586
validate_inputs() (*evalml.objectives.MedianAE method*), 588
validate_inputs() (*evalml.objectives.MSE method*), 590
validate_inputs() (*evalml.objectives.multiclass_classification_objective method*), 454
validate_inputs() (*evalml.objectives.MulticlassClassificationObjective method*), 592
validate_inputs() (*evalml.objectives.objective_base.ObjectiveBase method*), 456
validate_inputs() (*evalml.objectives.ObjectiveBase method*), 594
validate_inputs() (*evalml.objectives.Precision method*), 596
validate_inputs() (*evalml.objectives.PrecisionMacro method*), 598
validate_inputs() (*evalml.objectives.PrecisionMicro method*), 600
validate_inputs() (*evalml.objectives.PrecisionWeighted method*), 601
validate_inputs() (*evalml.objectives.R2 method*), 603
validate_inputs() (*evalml.objectives.Recall method*), 605
validate_inputs() (*evalml.objectives.RecallMacro method*), 607
validate_inputs() (*evalml.objectives.RecallMicro method*), 609
validate_inputs() (*evalml.objectives.RecallWeighted method*), 610
validate_inputs() (*evalml.objectives.regression_objective.RegressionObjective method*), 459
validate_inputs() (*evalml.objectives.RegressionObjective method*), 613

validate_inputs() (*evalml.objectives.RootMeanSquaredLogError method*), 614
validate_inputs() (*evalml.objectives.RootMeanSquaredLogError method*), 616
validate_inputs() (*evalml.objectives.sensitivity_low_alert.SensitivityLowAlert method*), 462
validate_inputs() (*evalml.objectives.SensitivityLowAlert method*), 618
validate_inputs() (*evalml.objectives.standard_metrics.AccuracyBinary method*), 465
validate_inputs() (*evalml.objectives.standard_metrics.AccuracyMulticlass method*), 467
validate_inputs() (*evalml.objectives.standard_metrics.AUC method*), 469
validate_inputs() (*evalml.objectives.standard_metrics.AUCMacro method*), 471
validate_inputs() (*evalml.objectives.standard_metrics.AUCMicro method*), 472
validate_inputs() (*evalml.objectives.standard_metrics.AUCWeighted method*), 474
validate_inputs() (*evalml.objectives.standard_metrics.BalancedAccuracyBinary method*), 476
validate_inputs() (*evalml.objectives.standard_metrics.BalancedAccuracyMulticlass method*), 478
validate_inputs() (*evalml.objectives.standard_metrics.EqualizedOdds method*), 480
validate_inputs() (*evalml.objectives.standard_metrics.F1 method*), 482
validate_inputs() (*evalml.objectives.standard_metrics.F1Macro method*), 484
validate_inputs() (*evalml.objectives.standard_metrics.F1Micro method*), 485
validate_inputs() (*evalml.objectives.standard_metrics.F1Weighted method*), 487
validate_inputs() (*evalml.objectives.standard_metrics.Gini method*), 489
validate_inputs() (*evalml.objectives.standard_metrics.LogLossBinary method*), 491
validate_inputs() (*evalml.objectives.standard_metrics.LogLossMulticlass method*), 493
validate_inputs() (*evalml.objectives.standard_metrics.MAE method*), 495
validate_inputs() (*evalml.objectives.standard_metrics.MAPE method*), 497
validate_inputs() (*evalml.objectives.standard_metrics.MaxError method*), 498
validate_inputs() (*evalml.objectives.standard_metrics.MCCBinary method*), 501
validate_inputs() (*evalml.objectives.standard_metrics.MCCMulticlass method*), 502
validate_inputs() (*evalml.objectives.standard_metrics.MeanSquaredLogError method*), 504
validate_inputs() (*evalml.objectives.standard_metrics.MedianAE method*), 506
validate_inputs() (*evalml.objectives.standard_metrics.MSE method*), 507
validate_inputs() (*evalml.objectives.standard_metrics.Precision method*), 510
validate_inputs() (*evalml.objectives.standard_metrics.PrecisionMacro method*), 511
validate_inputs() (*evalml.objectives.standard_metrics.PrecisionMicro method*), 513
validate_inputs() (*evalml.objectives.standard_metrics.PrecisionWeighted method*), 515
validate_inputs() (*evalml.objectives.standard_metrics.R2 method*), 516
validate_inputs() (*evalml.objectives.standard_metrics.Recall method*), 519
validate_inputs() (*evalml.objectives.standard_metrics.RecallMacro method*), 520
validate_inputs() (*evalml.objectives.standard_metrics.RecallMicro method*), 522
validate_inputs() (*evalml.objectives.standard_metrics.RecallWeighted method*), 524
validate_inputs() (*evalml.objectives.standard_metrics.RootMeanSquaredLogError method*), 525
validate_inputs() (*evalml.objectives.standard_metrics.RootMeanSquaredLogError method*), 527
validate_inputs() (*evalml.objectives.time_series_regression_objective.TimeSeriesRegressionObjective method*), 530
ValidationErrorCode (*class in evalml.exceptions*), 397
ValidationErrorCode (*class in evalml.exceptions.exceptions*), 394
value() (*evalml.data_checks.data_check_action_code.DataCheckActionCode method*), 291
value() (*evalml.data_checks.data_check_action_option.DCAOParameterAllowedValuesType method*), 293
value() (*evalml.data_checks.data_check_action_option.DCAOParameterType method*), 294
value() (*evalml.data_checks.data_check_message_code.DataCheckMessageCode method*), 298
value() (*evalml.data_checks.data_check_message_type.DataCheckMessageType method*), 298
value() (*evalml.data_checks.DataCheckActionCode method*), 346
value() (*evalml.data_checks.DataCheckMessageCode method*), 350
value() (*evalml.data_checks.DataCheckMessageType method*), 351
value() (*evalml.data_checks.DCAOParameterAllowedValuesType method*), 359
value() (*evalml.data_checks.DCAOParameterType method*), 360
value() (*evalml.exceptions.exceptions.PartialDependenceErrorCode method*), 393
value() (*evalml.exceptions.exceptions.PipelineErrorCodeEnum method*), 394
value() (*evalml.exceptions.exceptions.ValidationErrorCode*

method), 395
value() (evalml.exceptions.PartialDependenceErrorCode method), 396
value() (evalml.exceptions.PipelineErrorCodeEnum method), 397
value() (evalml.exceptions.ValidationErrorCode method), 398
value() (evalml.model_family.model_family.ModelFamily method), 399
value() (evalml.model_family.ModelFamily method), 401
value() (evalml.model_understanding.prediction_explanations.explainers.ExplainPredictionsStage method), 406
value() (evalml.problem_types.problem_types.ProblemTypes method), 1876
value() (evalml.problem_types.ProblemTypes method), 1883
visualize_decision_tree() (in module evalml.model_understanding.visualizations), 425
VowpalWabbitBaseClassifier (class in evalml.pipelines.components.estimators.classifiers.vowpal_wabbit_classifiers), 682
VowpalWabbitBinaryClassifier (class in evalml.pipelines), 1837
VowpalWabbitBinaryClassifier (class in evalml.pipelines.components), 1537
VowpalWabbitBinaryClassifier (class in evalml.pipelines.components.estimators), 941
VowpalWabbitBinaryClassifier (class in evalml.pipelines.components.estimators.classifiers), 730
VowpalWabbitBinaryClassifier (class in evalml.pipelines.components.estimators.classifiers.vowpal_wabbit_classifiers), 685
VowpalWabbitMulticlassClassifier (class in evalml.pipelines), 1840
VowpalWabbitMulticlassClassifier (class in evalml.pipelines.components), 1540
VowpalWabbitMulticlassClassifier (class in evalml.pipelines.components.estimators), 944
VowpalWabbitMulticlassClassifier (class in evalml.pipelines.components.estimators.classifiers), 734
VowpalWabbitMulticlassClassifier (class in evalml.pipelines.components.estimators.classifiers.vowpal_wabbit_classifiers), 688
VowpalWabbitRegressor (class in evalml.pipelines), 1843
VowpalWabbitRegressor (class in evalml.pipelines.components), 1543
VowpalWabbitRegressor (class in evalml.pipelines.components.estimators), 947
VowpalWabbitRegressor (class in evalml.pipelines.components.estimators.regressors), 846
VowpalWabbitRegressor (class in evalml.pipelines.components.estimators.regressors.vowpal_wabbi 793

## W

warning() (evalml.automl.engine.engine_base.JobLogger method), 246
warning_not_unique_enough (in module evalml.data_checks.uniqueness_data_check), 340
warning_too_unique (in module evalml.data_checks.sparsity_data_check), 330
warning_too_unique (in module evalml.data_checks.uniqueness_data_check), 340
WrappedSKClassifier (class in evalml.pipelines.components.utils), 1331
WrappedSKRegressor (class in evalml.pipelines.components.utils), 1332
write_to_logger() (evalml.automl.engine.engine_base.JobLogger method), 246

## X

XGBoostClassifier (class in evalml.pipelines), 1846
XGBoostClassifier (class in evalml.pipelines.components), 1546
XGBoostClassifier (class in evalml.pipelines.components.estimators), 950
XGBoostClassifier (class in evalml.pipelines.components.estimators.classifiers), 737
XGBoostClassifier (class in evalml.pipelines.components.estimators.classifiers.xgboost_classi 692
XGBoostRegressor (class in evalml.pipelines), 1849
XGBoostRegressor (class in evalml.pipelines.components), 1549
XGBoostRegressor (class in evalml.pipelines.components.estimators), 953
XGBoostRegressor (class in evalml.pipelines.components.estimators.regressors), 850
XGBoostRegressor (class in evalml.pipelines.components.estimators.regressors.xgboost_regre 797