EvalML Documentation

Release 0.48.0

Alteryx, Inc.

Mar 29, 2022

CONTENTS

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 3.7, 3.8, and 3.9. It can be installed from pypi, conda-forge, or from source.

To install EvalML with all dependencies, run the following command:

```
$ pip install evalml
```

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

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, and optional requirements are isted in `requirements.txt`.

To install EvalML with only the core-required dependencies with pypi, first download the EvalML source from pypi to access the requirements files before running the following command. For conda-forge, just run 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[update_checker]
```

```
$ 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. Prophet is currently only supported via pip installation in EvalML for Mac with CmdStan as a backend.

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

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

Note: In order to do this, you must have the EvalML repo cloned and you must be in the top level folder `<your_directory>/evalml/` to execute this command. This command will do the following:

- Pip install `cmdstanpy==0.9.68`
- Execute the `install_cmdstan.py` script found within your `site-packages/cmdstanpy` which builds `cmdstan` in your `site-packages`.
- Install `Prophet==1.0.1` with the `CMDSTAN` and `STAN_BACKEND` environment variables set.

If the `site-packages` path is incorrect or you'd like to specify a different one, just run `make installdeps-prophet SITE_PACKAGES_DIR="<path_to_your_site_packages>"`.

If you'd like to have more fine-tuned control over the installation steps for Prophet, such as specifying the backend, follow these steps:

```
$ pip install prophet==1.0.1
```

```
$ pip install cmdstanpy==0.9.68
$ python <path_to_installed_cmdstanpy>/install_cmdstan.py --dir <path_to_build_cmdstan> -
↪v <version_to_use>
$ CMDSTAN=<path_to_build_cmdstan>/cmdstan-<version_to_use> STAN_BACKEND=CMDSTANPY pip␣
↪install prophet==1.0.1
```

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
```

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= {'store_id': 'categorical',
                                           'expiration_date': 'datetime',
                                           'lat': 'categorical',
                                           'lng': 'categorical',
```

(continues on next page)

(continued from previous page)

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

```
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()
```

```
*****************************
* 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:
```

```
FigureWidget({
    'data': [{'mode': 'lines+markers',
              'name': 'Best Score',
              'type'...
```

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

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 F1: 0.227
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 *
*****************************

Logistic Regression Classifier w/ Label Encoder + Drop Columns Transformer + DateTime␣
↪Featurizer + Imputer + One Hot Encoder + Oversampler + Standard Scaler + RF Classifier␣
↪Select From Model:
        Starting cross validation
        Finished cross validation - mean F1: 0.276
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 + Oversampler +␣
↪Select Columns Transformer + Select Columns Transformer + Label Encoder + Imputer +␣
↪One Hot Encoder + Oversampler:
        Starting cross validation
        Finished cross validation - mean F1: 0.297
LightGBM Classifier w/ Label Encoder + Select Columns By Type Transformer + Label␣
↪Encoder + Drop Columns Transformer + DateTime Featurizer + Imputer + Oversampler +␣
↪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 + Oversampler +␣
↪Select Columns Transformer + Select Columns Transformer + Label Encoder + Imputer +␣
↪One Hot Encoder + Oversampler:
        Starting cross validation
        Finished cross validation - mean F1: 0.365
Elastic Net Classifier w/ Label Encoder + Select Columns By Type Transformer + Label␣
↪Encoder + Drop Columns Transformer + DateTime Featurizer + Imputer + Oversampler +␣
↪Standard Scaler + Select Columns Transformer + Select Columns Transformer + Label␣
↪Encoder + Imputer + One Hot Encoder + Oversampler + Standard Scaler:
        Starting cross validation
        Finished cross validation - mean F1: 0.235
CatBoost Classifier w/ Label Encoder + Select Columns By Type Transformer + Label␣
↪Encoder + Drop Columns Transformer + DateTime Featurizer + Imputer + Oversampler +␣
↪Select Columns Transformer + Select Columns Transformer + Label Encoder + Imputer +␣
↪Oversampler:
        Starting cross validation
        Finished cross validation - mean F1: 0.683
XGBoost Classifier w/ Label Encoder + Select Columns By Type Transformer + Label Encoder␣
↪+ Drop Columns Transformer + DateTime Featurizer + Imputer + Oversampler + Select␣
↪Columns Transformer + Select Columns Transformer + Label Encoder + Imputer + One Hot␣
↪Encoder + Oversampler:
        Starting cross validation
        Finished cross validation - mean F1: 0.690

Search finished after 00:27
Best pipeline: XGBoost Classifier w/ Label Encoder + Select Columns By Type Transformer␣
↪+ Label Encoder + Drop Columns Transformer + DateTime Featurizer + Imputer +␣
↪Oversampler + Select Columns Transformer + Select Columns Transformer + Label Encoder␣
↪+ Imputer + One Hot Encoder + Oversampler
Best pipeline F1: 0.689744
```

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

[7]:
```
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=False)
automl.search()
```

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   10  XGBoost Classifier w/ Label Encoder + Select C...            10
1    9  CatBoost Classifier w/ Label Encoder + Select ...             9
2    4  Random Forest Classifier w/ Label Encoder + Dr...             4
3    6  LightGBM Classifier w/ Label Encoder + Select ...             6
4    2  Random Forest Classifier w/ Label Encoder + Dr...             2
5    7  Extra Trees Classifier w/ Label Encoder + Sele...             7
6    5  Decision Tree Classifier w/ Label Encoder + Se...             5
7    3  Logistic Regression Classifier w/ Label Encode...             3
8    8  Elastic Net Classifier w/ Label Encoder + Sele...             8
9    1  Logistic Regression Classifier w/ Label Encode...             1
10   0       Mode Baseline Binary Classification Pipeline             0

    mean_cv_score  standard_deviation_cv_score  validation_score  \
0        0.689744                     0.165041          0.689744
1        0.682540                     0.168919          0.682540
2        0.663337                     0.263244          0.663337
3        0.588889                     0.083887          0.588889
4        0.558405                     0.182781          0.558405
5        0.364957                     0.085286          0.364957
6        0.296626                     0.114974          0.296626
7        0.275556                     0.120247          0.275556
8        0.235240                     0.063793          0.235240
9        0.227085                     0.072648          0.227085
10       0.000000                     0.000000          0.000000

    percent_better_than_baseline  high_variance_cv  \
0                      68.974359             False
1                      68.253968             False
2                      66.333666             False
3                      58.888889             False
4                      55.840456             False
5                      36.495726             False
6                      29.662639             False
7                      27.555556             False
8                      23.523954             False
9                      22.708518             False
10                      0.000000             False
```

(continues on next page)

(continued from previous page)

```
                                              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}, 'D...
8   {'Label Encoder': {'positive_label': None}, 'N...
9   {'Label Encoder': {'positive_label': None}, 'D...
10  {'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)
```

```
*************************************************************************************************************
* Logistic Regression Classifier w/ Label Encoder + Drop Columns Transformer + DateTime␣
↪Featurizer + Imputer + One Hot Encoder + Oversampler + Standard Scaler + RF Classifier␣
↪Select From Model *
*************************************************************************************************************

Problem Type: binary
Model Family: Linear

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
         * categorical_fill_value : None
         * numeric_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
```

(continues on next page)

(continued from previous page)

```
        * n_jobs : -1
        * sampling_ratio_dict : None
        * categorical_features : [3, 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]
        * k_neighbors : 5
7. Standard Scaler
8. RF Classifier Select From Model
        * number_features : None
        * n_estimators : 10
        * max_depth : None
        * percent_features : 0.5
        * threshold : median
        * n_jobs : -1
9. Logistic Regression Classifier
        * penalty : l2
        * C : 1.0
        * n_jobs : -1
        * multi_class : auto
        * solver : lbfgs

Training
========
Training for binary problems.
Objective to optimize binary classification pipeline thresholds for: <evalml.objectives.
↪standard_metrics.F1 object at 0x7f6ee918c610>
Total training time (including CV): 2.6 seconds

Cross Validation
----------------
              F1  MCC Binary  Log Loss Binary   Gini   AUC  Precision  Balanced␣
↪Accuracy Binary  Accuracy Binary # Training # Validation
0           0.400       0.308            0.438  0.288 0.644      0.333                ␣
↪  0.682            0.821        133           67
1           0.267       0.175            0.609  0.208 0.604      0.286                ␣
↪  0.583            0.836        133           67
2           0.160       0.010            0.673 -0.017 0.492      0.111                ␣
↪  0.507            0.682        134           66
mean        0.276       0.164            0.573  0.160 0.580      0.243                ␣
↪  0.591            0.780          -            -
std         0.120       0.149            0.121  0.158 0.079      0.117                ␣
↪  0.088            0.085          -            -
coef of var 0.436       0.908            0.212  0.991 0.136      0.481                ␣
↪  0.149            0.109          -            -
```

We can also view the pipeline parameters directly:

[10]:
```
pipeline = automl.get_pipeline(3)
print(pipeline.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',
↪'categorical_fill_value': None, 'numeric_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, 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], 'k_neighbors': 5}, 'RF Classifier␣
↪Select From Model': {'number_features': None, 'n_estimators': 10, 'max_depth': None,
```

(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=.5,
                            interchange_fee=.02,
                            fraud_payout_percentage=.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]:

Column	Physical Type	Logical Type	Semantic Tag(s)
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)

automl.search()
```

```
*****************************
* 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:
```

```
FigureWidget({
    'data': [{'mode': 'lines+markers',
              'name': 'Best Score',
              'type'...
```

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

Logistic Regression Classifier w/ Label Encoder + Imputer + One Hot Encoder +␣
↪Oversampler + Standard Scaler:
        Starting cross validation
        Finished cross validation - mean Fraud Cost: 0.008
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 00:08
Best pipeline: Logistic Regression Classifier w/ Label Encoder + Imputer + One Hot␣
↪Encoder + Oversampler + Standard Scaler
Best pipeline Fraud Cost: 0.008198
```

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  Logistic Regression Classifier w/ Label Encode...             1
1   2  Random Forest Classifier w/ Label Encoder + Im...             2
2   0       Mode Baseline Binary Classification Pipeline             0

   mean_cv_score  standard_deviation_cv_score  validation_score  \
```

(continues on next page)

(continued from previous page)

```
0       0.008198                    0.000424          0.008198
1       0.008745                    0.001207          0.008745
2       0.789648                    0.001136          0.789648

   percent_better_than_baseline  high_variance_cv  \
0                     78.145054             False
1                     78.090334             False
2                      0.000000             False

                                          parameters
0  {'Label Encoder': {'positive_label': None}, 'I...
1  {'Label Encoder': {'positive_label': None}, 'I...
2  {'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"])
```

```
*********************************************************************************************
* Random Forest Classifier w/ Label Encoder + Imputer + One Hot Encoder + Oversampler *
*********************************************************************************************

Problem Type: binary
Model Family: Random Forest

Pipeline Steps
==============
1. Label Encoder
         * positive_label : None
2. Imputer
         * categorical_impute_strategy : most_frequent
         * numeric_impute_strategy : mean
         * categorical_fill_value : None
         * numeric_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. 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 : [3, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15]
         * k_neighbors : 5
5. Random Forest Classifier
         * n_estimators : 100
         * max_depth : 6
         * n_jobs : -1

Training
========
Training for binary problems.
Objective to optimize binary classification pipeline thresholds for: <evalml.objectives.
↪fraud_cost.FraudCost object at 0x7f52a65f9cd0>
Total training time (including CV): 2.9 seconds

Cross Validation
----------------
             Fraud Cost    AUC     F1  Precision # Training # Validation
0                 0.008 0.864 0.241      0.137      2,666        1,334
1                 0.010 0.852 0.244      0.139      2,667        1,333
2                 0.008 0.844 0.244      0.139      2,667        1,333
mean              0.009 0.853 0.243      0.138          -            -
std               0.001 0.010 0.001      0.001          -            -
coef of var       0.138 0.012 0.005      0.006          -            -
```

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', 0.8096439019469384),
             ('Fraud Cost', 0.008828859021444526)])
```

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,
```

(continues on next page)

(continued from previous page)

```
                            verbose=True)

automl_auc.search()
```

```
*****************************
* 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:
```

```
FigureWidget({
    'data': [{'mode': 'lines+markers',
              'name': 'Best Score',
              'type'...
```

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

Logistic Regression Classifier w/ Label Encoder + Imputer + One Hot Encoder +␣
↪Oversampler + Standard Scaler:
        Starting cross validation
        Finished cross validation - mean AUC: 0.827
Random Forest Classifier w/ Label Encoder + Imputer + One Hot Encoder + Oversampler:
        Starting cross validation
        Finished cross validation - mean AUC: 0.853

Search finished after 00:05
Best pipeline: Random Forest Classifier w/ Label Encoder + Imputer + One Hot Encoder +␣
↪Oversampler
Best pipeline AUC: 0.853164
```

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   2  Random Forest Classifier w/ Label Encoder + Im...             2
1   1  Logistic Regression Classifier w/ Label Encode...             1
2   0       Mode Baseline Binary Classification Pipeline             0

   mean_cv_score  standard_deviation_cv_score  validation_score  \
```

(continues on next page)

(continued from previous page)

```
0       0.853164                    0.010011          0.853164
1       0.826675                    0.027444          0.826675
2       0.500000                    0.000000          0.500000

   percent_better_than_baseline  high_variance_cv  \
0                     35.316397             False
1                     32.667497             False
2                      0.000000             False

                                          parameters
0  {'Label Encoder': {'positive_label': None}, 'I...
1  {'Label Encoder': {'positive_label': None}, 'I...
2  {'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', 0.8096439019469384),
             ('Fraud Cost', 0.008828859021444526)])
```

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('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          float64       Double     ['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          float64       Double     ['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()
```

```
*****************************
* 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.
```

(continues on next page)

(continued from previous page)

```
Allowed model families:
```

```
FigureWidget({
    'data': [{'mode': 'lines+markers',
              'name': 'Best Score',
              'type'...
```

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

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

Logistic Regression Classifier w/ Label Encoder + Imputer + One Hot Encoder +␣
↪Oversampler + Standard Scaler:
        Starting cross validation
        Finished cross validation - mean Lead Scoring: -0.004
Random Forest Classifier w/ Label Encoder + Imputer + One Hot Encoder + Oversampler:
        Starting cross validation
        Finished cross validation - mean Lead Scoring: 0.039

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

Logistic Regression Classifier w/ Label Encoder + Imputer + One Hot Encoder +␣
↪Oversampler + Standard Scaler + RF Classifier Select From Model:
        Starting cross validation
        Finished cross validation - mean Lead Scoring: 0.041
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: 0.016

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

Decision Tree Classifier w/ Label Encoder + Select Columns By Type Transformer + Label␣
↪Encoder + Imputer + Oversampler + Select Columns Transformer + Select Columns␣
↪Transformer + Label Encoder + Imputer + One Hot Encoder + Oversampler:
        Starting cross validation
        Finished cross validation - mean Lead Scoring: 0.011
LightGBM Classifier w/ Label Encoder + Select Columns By Type Transformer + Label␣
↪Encoder + Imputer + Oversampler + Select Columns Transformer + Select Columns␣
↪Transformer + Label Encoder + Imputer + One Hot Encoder + Oversampler:
        Starting cross validation
        Finished cross validation - mean Lead Scoring: -0.010
Extra Trees Classifier w/ Label Encoder + Select Columns By Type Transformer + Label␣
↪Encoder + Imputer + Oversampler + 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 Lead Scoring: 0.009
Elastic Net Classifier w/ Label Encoder + Select Columns By Type Transformer + Label␣
↪Encoder + Imputer + Oversampler + Standard Scaler + Select Columns Transformer +␣
↪Select Columns Transformer + Label Encoder + Imputer + One Hot Encoder + Oversampler +␣
↪Standard Scaler:
        Starting cross validation
        Finished cross validation - mean Lead Scoring: 0.000
CatBoost Classifier w/ Label Encoder + Select Columns By Type Transformer + Label␣
↪Encoder + Imputer + Oversampler + Select Columns Transformer + Select Columns␣
↪Transformer + Label Encoder + Imputer + Oversampler:
        Starting cross validation
        Finished cross validation - mean Lead Scoring: 0.360
XGBoost Classifier w/ Label Encoder + Select Columns By Type Transformer + Label Encoder␣
↪+ Imputer + Oversampler + Select Columns Transformer + Select Columns Transformer +␣
↪Label Encoder + Imputer + One Hot Encoder + Oversampler:
        Starting cross validation
        Finished cross validation - mean Lead Scoring: 0.004

Search finished after 00:28
Best pipeline: CatBoost Classifier w/ Label Encoder + Select Columns By Type Transformer␣
↪+ Label Encoder + Imputer + Oversampler + Select Columns Transformer + Select Columns␣
↪Transformer + Label Encoder + Imputer + Oversampler
Best pipeline Lead Scoring: 0.360294
```

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	mean_cv_score	standard_deviation_cv_score	validation_score
0	9	CatBoost Classifier w/ Label Encoder + Select ...	9	0.360294	0.072885	0.360294
1	3	Logistic Regression Classifier w/ Label Encode...	3	0.040866	0.065818	0.040866
2	2	Random Forest Classifier w/ Label Encoder + Im...	2	0.038720	0.034897	0.038720
3	4	Random Forest Classifier w/ Label Encoder + Im...	4	0.016129	0.027936	0.016129
4	5	Decision Tree Classifier w/ Label Encoder + Se...	5	0.010753	0.018624	0.010753
5	7	Extra Trees Classifier w/ Label Encoder + Sele...	7			
6	10	XGBoost Classifier w/ Label Encoder + Select C...	10			
7	0	Mode Baseline Binary Classification Pipeline	0			
8	8	Elastic Net Classifier w/ Label Encoder + Sele...	8			
9	1	Logistic Regression Classifier w/ Label Encode...	1			
10	6	LightGBM Classifier w/ Label Encoder + Select ...	6			

(continues on next page)

(continued from previous page)

```
5        0.008602                    0.014899          0.008602
6        0.004301                    0.007450          0.004301
7        0.000000                    0.000000          0.000000
8        0.000000                    0.000000          0.000000
9       -0.004300                    0.009857         -0.004300
10      -0.009681                    0.008539         -0.009681

    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
10                           inf             False

                                          parameters
0   {'Label Encoder': {'positive_label': None}, 'N...
1   {'Label Encoder': {'positive_label': None}, 'I...
2   {'Label Encoder': {'positive_label': None}, 'I...
3   {'Label Encoder': {'positive_label': None}, 'I...
4   {'Label Encoder': {'positive_label': None}, 'N...
5   {'Label Encoder': {'positive_label': None}, 'N...
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}, 'I...
10  {'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 + Oversampler + Select Columns Transformer + Select Columns␣
↪Transformer + Label Encoder + Imputer + Oversampler *
*****************************************************************************************************************

Problem Type: binary
```

(continues on next page)

(continued from previous page)

```
Model Family: CatBoost

Pipeline Steps
==============
1. Label Encoder
         * positive_label : None
2. Select Columns By Type Transformer
         * column_types : ['category']
         * exclude : True
3. Label Encoder
         * positive_label : None
4. Imputer
         * categorical_impute_strategy : most_frequent
         * numeric_impute_strategy : mean
         * categorical_fill_value : None
         * numeric_fill_value : None
5. Oversampler
         * sampling_ratio : 0.25
         * k_neighbors_default : 5
         * n_jobs : -1
         * sampling_ratio_dict : None
         * k_neighbors : 5
6. Select Columns Transformer
         * columns : ['zip', 'amount']
7. Select Columns Transformer
         * columns : ['job', 'state', 'action']
8. Label Encoder
         * positive_label : None
9. Imputer
         * categorical_impute_strategy : most_frequent
         * numeric_impute_strategy : mean
         * categorical_fill_value : None
         * numeric_fill_value : None
10. Oversampler
         * sampling_ratio : 0.25
         * k_neighbors_default : 5
         * n_jobs : -1
         * sampling_ratio_dict : None
         * k_neighbors : 5
11. 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 0x7f70a6ee5bb0>
```

(continues on next page)

(continued from previous page)

```
Total training time (including CV): 1.2 seconds

Cross Validation
----------------
             Lead Scoring   AUC # Training # Validation
0                   0.429 0.846      3,099        1,550
1                   0.284 0.890      3,099        1,550
2                   0.368 0.891      3,100        1,549
mean                0.360 0.876          -            -
std                 0.073 0.025          -            -
coef of var         0.202 0.029          -            -
```

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', 0.8733152009670595),
             ('Lead Scoring', 0.5030094582975064)])
```

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()
```

```
*****************************
* 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.
```

(continues on next page)

(continued from previous page)

```
Allowed model families:
```

```
FigureWidget({
    'data': [{'mode': 'lines+markers',
              'name': 'Best Score',
              'type'...
```

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

Logistic Regression Classifier w/ Label Encoder + Imputer + One Hot Encoder +␣
↪Oversampler + Standard Scaler:
        Starting cross validation
        Finished cross validation - mean AUC: 0.653
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 *
*****************************

Logistic Regression Classifier w/ Label Encoder + Imputer + One Hot Encoder +␣
↪Oversampler + Standard Scaler + RF Classifier Select From Model:
        Starting cross validation
        Finished cross validation - mean AUC: 0.643
Random Forest Classifier w/ Label Encoder + Imputer + One Hot Encoder + Oversampler + RF␣
↪Classifier Select From Model:
        Starting cross validation
        Finished cross validation - mean AUC: 0.637

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

Decision Tree Classifier w/ Label Encoder + Select Columns By Type Transformer + Label␣
↪Encoder + Imputer + Oversampler + Select Columns Transformer + Select Columns␣
↪Transformer + Label Encoder + Imputer + One Hot Encoder + Oversampler:
        Starting cross validation
        Finished cross validation - mean AUC: 0.579
LightGBM Classifier w/ Label Encoder + Select Columns By Type Transformer + Label␣
↪Encoder + Imputer + Oversampler + Select Columns Transformer + Select Columns␣
↪Transformer + Label Encoder + Imputer + One Hot Encoder + Oversampler:
        Starting cross validation
        Finished cross validation - mean AUC: 0.601
Extra Trees Classifier w/ Label Encoder + Select Columns By Type Transformer + Label␣
↪Encoder + Imputer + Oversampler + 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 AUC: 0.649
Elastic Net Classifier w/ Label Encoder + Select Columns By Type Transformer + Label␣
↪Encoder + Imputer + Oversampler + Standard Scaler + Select Columns Transformer +␣
↪Select Columns Transformer + Label Encoder + Imputer + One Hot Encoder + Oversampler +␣
↪Standard Scaler:
        Starting cross validation
        Finished cross validation - mean AUC: 0.635
CatBoost Classifier w/ Label Encoder + Select Columns By Type Transformer + Label␣
↪Encoder + Imputer + Oversampler + Select Columns Transformer + Select Columns␣
↪Transformer + Label Encoder + Imputer + Oversampler:
        Starting cross validation
        Finished cross validation - mean AUC: 0.876
XGBoost Classifier w/ Label Encoder + Select Columns By Type Transformer + Label Encoder␣
↪+ Imputer + Oversampler + Select Columns Transformer + Select Columns Transformer +␣
↪Label Encoder + Imputer + One Hot Encoder + Oversampler:
        Starting cross validation
        Finished cross validation - mean AUC: 0.628

Search finished after 00:29
Best pipeline: CatBoost Classifier w/ Label Encoder + Select Columns By Type Transformer␣
↪+ Label Encoder + Imputer + Oversampler + Select Columns Transformer + Select Columns␣
↪Transformer + Label Encoder + Imputer + Oversampler
Best pipeline AUC: 0.875697
```

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

[12]:
```
    id                                     pipeline_name  search_order  \
0    9  CatBoost Classifier w/ Label Encoder + Select ...             9
1    1  Logistic Regression Classifier w/ Label Encode...             1
2    2  Random Forest Classifier w/ Label Encoder + Im...             2
3    7  Extra Trees Classifier w/ Label Encoder + Sele...             7
4    3  Logistic Regression Classifier w/ Label Encode...             3
5    4  Random Forest Classifier w/ Label Encoder + Im...             4
6    8  Elastic Net Classifier w/ Label Encoder + Sele...             8
7   10  XGBoost Classifier w/ Label Encoder + Select C...            10
8    6  LightGBM Classifier w/ Label Encoder + Select ...             6
9    5  Decision Tree Classifier w/ Label Encoder + Se...             5
10   0       Mode Baseline Binary Classification Pipeline             0

    mean_cv_score  standard_deviation_cv_score  validation_score  \
0        0.875697                     0.025383          0.875697
1        0.652954                     0.036080          0.652954
2        0.651591                     0.059622          0.651591
3        0.649083                     0.036805          0.649083
4        0.642721                     0.050740          0.642721
5        0.637190                     0.049395          0.637190
6        0.634544                     0.024862          0.634544
7        0.628339                     0.018670          0.628339
8        0.600536                     0.011945          0.600536
9        0.579447                     0.010766          0.579447
10       0.500000                     0.000000          0.500000
```

(continues on next page)

(continued from previous page)

```
    percent_better_than_baseline  high_variance_cv  \
0                      37.569667             False
1                      15.295408             False
2                      15.159098             False
3                      14.908343             False
4                      14.272080             False
5                      13.719027             False
6                      13.454355             False
7                      12.833910             False
8                      10.053587             False
9                       7.944728             False
10                      0.000000             False

                                           parameters
0   {'Label Encoder': {'positive_label': None}, 'N...
1   {'Label Encoder': {'positive_label': None}, 'I...
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}, 'I...
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}, 'N...
10  {'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', 0.8733152009670595),
             ('Lead Scoring', 0.2880481513327601)])
```

[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()
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()

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

```
Removing columns ['TotalCharges'] because they are of 'Unknown' type

*****************************
* 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:
```

```
FigureWidget({
    'data': [{'mode': 'lines+markers',
              'name': 'Best Score',
              'type'...
```

```
Evaluating Baseline Pipeline: Mode Baseline Binary Classification Pipeline
Mode Baseline Binary Classification Pipeline:
        Starting cross validation
        Finished cross validation - mean Log Loss Binary: 9.164

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

Logistic Regression Classifier w/ Label Encoder + Drop Columns Transformer + Imputer +␣
↪One Hot Encoder + Standard Scaler:
        Starting cross validation
        Finished cross validation - mean Log Loss Binary: 0.423
Random Forest Classifier w/ Label Encoder + Drop Columns Transformer + Imputer + One Hot␣
↪Encoder:
        Starting cross validation
        Finished cross validation - mean Log Loss Binary: 0.426

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

Logistic Regression Classifier w/ Label Encoder + Drop Columns Transformer + Imputer +␣
↪One Hot Encoder + Standard Scaler + RF Classifier Select From Model:
        Starting cross validation
        Finished cross validation - mean Log Loss Binary: 0.423
Random Forest Classifier w/ Label Encoder + Drop Columns Transformer + Imputer + One Hot␣
↪Encoder + RF Classifier Select From Model:
        Starting cross validation
        Finished cross validation - mean Log Loss Binary: 0.426

Search finished after 00:13
Best pipeline: Logistic Regression Classifier w/ Label Encoder + Drop Columns␣
↪Transformer + Imputer + One Hot Encoder + Standard Scaler
Best pipeline Log Loss Binary: 0.422791
```

[3]:
```
OrderedDict([('Log Loss Binary', 0.4199275026717707)])
```

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', 52.38996687174633)])
```

[5]:
```
# Calculate total profit across all customers using pipeline optimized for Log Loss
```

(continues on next page)

(continued from previous page)

```
total_profit_ll = ll_pipeline_score['Cost Benefit Matrix'] * len(X)
print (total_profit_ll)
```

```
368982.53667770943
```

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()

cbm_pipeline = automl.best_pipeline
```

```
Removing columns ['TotalCharges'] because they are of 'Unknown' type

*****************************
* 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:
```

```
FigureWidget({
    'data': [{'mode': 'lines+markers',
              'name': 'Best Score',
              'type'...
```

```
Evaluating Baseline Pipeline: Mode Baseline Binary Classification Pipeline
Mode Baseline Binary Classification Pipeline:
        Starting cross validation
        Finished cross validation - mean Cost Benefit Matrix: -53.063

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

Logistic Regression Classifier w/ Label Encoder + Drop Columns Transformer + Imputer +␣
↪One Hot Encoder + Standard Scaler:
        Starting cross validation
        Finished cross validation - mean Cost Benefit Matrix: 59.696
Random Forest Classifier w/ Label Encoder + Drop Columns Transformer + Imputer + One Hot␣
↪Encoder:
        Starting cross validation
        Finished cross validation - mean Cost Benefit Matrix: 59.392
```

(continues on next page)

(continued from previous page)

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

Logistic Regression Classifier w/ Label Encoder + Drop Columns Transformer + Imputer +␣
↪One Hot Encoder + Standard Scaler + RF Classifier Select From Model:
        Starting cross validation
        Finished cross validation - mean Cost Benefit Matrix: 58.925
Random Forest Classifier w/ Label Encoder + Drop Columns Transformer + Imputer + One Hot␣
↪Encoder + RF Classifier Select From Model:
        Starting cross validation
        Finished cross validation - mean Cost Benefit Matrix: 57.343

Search finished after 00:14
Best pipeline: Logistic Regression Classifier w/ Label Encoder + Drop Columns␣
↪Transformer + Imputer + One Hot Encoder + Standard Scaler
Best pipeline Cost Benefit Matrix: 59.696177
```

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', 62.13913866540464)])
```

[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)
```

```
437645.95362044487
```

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

```
68663.41694273544
```

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.graphs 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()
```

```
*****************************
* 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:
```

```
FigureWidget({
    'data': [{'mode': 'lines+markers',
```

(continues on next page)

(continued from previous page)

```
             'name': 'Best Score',
             'type'...
```

```
Evaluating Baseline Pipeline: Mode Baseline Binary Classification Pipeline
Mode Baseline Binary Classification Pipeline:
        Starting cross validation
        Finished cross validation - mean Log Loss Binary: 14.046

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

Logistic Regression Classifier w/ Label Encoder + Natural Language Featurizer + Imputer␣
↪+ Standard Scaler:
        Starting cross validation
        Finished cross validation - mean Log Loss Binary: 0.324
Random Forest Classifier w/ Label Encoder + Natural Language Featurizer + Imputer:
        Starting cross validation
        Finished cross validation - mean Log Loss Binary: 0.209

Search finished after 00:10
Best pipeline: Random Forest Classifier w/ Label Encoder + Natural Language Featurizer +␣
↪Imputer
Best pipeline Log Loss Binary: 0.209360
```

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   2  Random Forest Classifier w/ Label Encoder + Na...             2
1   1  Logistic Regression Classifier w/ Label Encode...             1
2   0       Mode Baseline Binary Classification Pipeline             0

   mean_cv_score  standard_deviation_cv_score  validation_score  \
0       0.209360                     0.040523          0.209360
1       0.324151                     0.096452          0.324151
2      14.045769                     0.099705         14.045769

   percent_better_than_baseline  high_variance_cv  \
0                     98.509448             False
1                     97.692183             False
2                      0.000000             False

                                      parameters
0  {'Label Encoder': {'positive_label': None}, 'I...
1  {'Label Encoder': {'positive_label': None}, 'I...
2  {'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
         * categorical_fill_value : None
         * numeric_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): 3.5 seconds

Cross Validation
----------------
             Log Loss Binary  MCC Binary  Gini   AUC  Precision    F1  Balanced Accuracy␣
↪Binary  Accuracy Binary # Training # Validation
0                      0.206       0.876 0.953 0.977      0.916 0.927                    ␣
↪ 0.940            0.940        400          200
1                      0.171       0.855 0.977 0.988      0.947 0.910                    ␣
↪ 0.921            0.930        400          200
2                      0.252       0.783 0.924 0.962      0.877 0.871                    ␣
↪ 0.891            0.895        400          200
mean                   0.209       0.838 0.951 0.976      0.913 0.903                    ␣
↪ 0.917            0.922          -            -
std                    0.041       0.049 0.026 0.013      0.035 0.029                    ␣
↪ 0.025            0.024          -            -
coef of var            0.194       0.059 0.028 0.014      0.038 0.032                    ␣
↪ 0.027            0.026          -            -
```

[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 core objectives for binary classification problems.

[12]:
```
scores = best_pipeline.score(X_holdout, y_holdout,  objectives=evalml.objectives.get_
↪core_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]'],
```

(continues on next page)

(continued from previous page)

```
 '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                 5.0                    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
```

(continues on next page)

(continued from previous page)

```
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()
```

```

*****************************
* 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:
```

```
FigureWidget({
    'data': [{'mode': 'lines+markers',
              'name': 'Best Score',
              'type'...
```

```
Evaluating Baseline Pipeline: Mode Baseline Binary Classification Pipeline
Mode Baseline Binary Classification Pipeline:
        Starting cross validation
        Finished cross validation - mean Log Loss Binary: 14.046

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

(continues on next page)

(continued from previous page)

```
Logistic Regression Classifier w/ Label Encoder + Natural Language Featurizer + Imputer␣
↪+ Standard Scaler:
        Starting cross validation
        Finished cross validation - mean Log Loss Binary: 0.324
Random Forest Classifier w/ Label Encoder + Natural Language Featurizer + Imputer:
        Starting cross validation
        Finished cross validation - mean Log Loss Binary: 0.209

Search finished after 00:06
Best pipeline: Random Forest Classifier w/ Label Encoder + Natural Language Featurizer +␣
↪Imputer
Best pipeline Log Loss Binary: 0.209360
```

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   2  Random Forest Classifier w/ Label Encoder + Na...             2
1   1  Logistic Regression Classifier w/ Label Encode...             1
2   0       Mode Baseline Binary Classification Pipeline             0

   mean_cv_score  standard_deviation_cv_score  validation_score  \
0       0.209360                     0.040523          0.209360
1       0.324151                     0.096452          0.324151
2      14.045769                     0.099705         14.045769

   percent_better_than_baseline  high_variance_cv  \
0                     98.509448             False
1                     97.692183             False
2                      0.000000             False

                                        parameters
0  {'Label Encoder': {'positive_label': None}, 'I...
1  {'Label Encoder': {'positive_label': None}, 'I...
2  {'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
```

(continues on next page)

(continued from previous page)

```
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
         * categorical_fill_value : None
         * numeric_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): 3.5 seconds

Cross Validation
----------------
             Log Loss Binary  MCC Binary  Gini   AUC  Precision    F1  Balanced Accuracy␣
↪Binary  Accuracy Binary # Training # Validation
0                      0.206       0.876 0.953 0.977      0.916 0.927                  ␣
↪ 0.940            0.940        400          200
1                      0.171       0.855 0.977 0.988      0.947 0.910                  ␣
↪ 0.921            0.930        400          200
2                      0.252       0.783 0.924 0.962      0.877 0.871                  ␣
↪ 0.891            0.895        400          200
mean                   0.209       0.838 0.951 0.976      0.913 0.903                  ␣
↪ 0.917            0.922          -            -
std                    0.041       0.049 0.026 0.013      0.035 0.029                  ␣
↪ 0.025            0.024          -            -
coef of var            0.194       0.059 0.028 0.014      0.038 0.032                  ␣
↪ 0.027            0.026          -            -
```

[23]:
```
# get standard performance metrics on holdout data
scores = best_pipeline_no_text.score(X_holdout, y_holdout, objectives=evalml.objectives.
↪get_core_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=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
```

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'})
```

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=.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.

[5]:
```
automl = evalml.automl.AutoMLSearch(X_train=X_train, y_train=y_train, problem_type=
↪'binary', verbose=True)
automl.search()
```

```
Removing columns ['currency'] because they are of 'Unknown' type
Using default limit of max_batches=4.


*****************************
* 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:
```

```
FigureWidget({
    'data': [{'mode': 'lines+markers',
              'name': 'Best Score',
              'type'...
```

```
Evaluating Baseline Pipeline: Mode Baseline Binary Classification Pipeline
Mode Baseline Binary Classification Pipeline:
        Starting cross validation
        Finished cross validation - mean Log Loss Binary: 3.970

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

Logistic Regression Classifier w/ Label Encoder + Drop Columns Transformer + DateTime␣
↪Featurizer + Imputer + One Hot Encoder + Oversampler + Standard Scaler:
        Starting cross validation
```

(continues on next page)

(continued from previous page)

```
        Finished cross validation - mean Log Loss Binary: 0.620
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: 0.292

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

Logistic Regression Classifier w/ Label Encoder + Drop Columns Transformer + DateTime␣
↪Featurizer + Imputer + One Hot Encoder + Oversampler + Standard Scaler + RF Classifier␣
↪Select From Model:
        Starting cross validation
        Finished cross validation - mean Log Loss Binary: 0.573
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: 0.289

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

Decision Tree Classifier w/ Label Encoder + Select Columns By Type Transformer + Label␣
↪Encoder + Drop Columns Transformer + DateTime Featurizer + Imputer + Oversampler +␣
↪Select Columns Transformer + Select Columns Transformer + Label Encoder + Imputer +␣
↪One Hot Encoder + Oversampler:
        Starting cross validation
        Finished cross validation - mean Log Loss Binary: 3.656
        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 + Oversampler␣
↪+ 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 + Oversampler +␣
↪Select Columns Transformer + Select Columns Transformer + Label Encoder + Imputer +␣
↪One Hot Encoder + Oversampler:
        Starting cross validation
        Finished cross validation - mean Log Loss Binary: 0.316
Extra Trees Classifier w/ Label Encoder + Select Columns By Type Transformer + Label␣
↪Encoder + Drop Columns Transformer + DateTime Featurizer + Imputer + Oversampler +␣
↪Select Columns Transformer + Select Columns Transformer + Label Encoder + Imputer +␣
↪One Hot Encoder + Oversampler:
        Starting cross validation
        Finished cross validation - mean Log Loss Binary: 0.344
Elastic Net Classifier w/ Label Encoder + Select Columns By Type Transformer + Label␣
↪Encoder + Drop Columns Transformer + DateTime Featurizer + Imputer + Oversampler +␣
↪Standard Scaler + Select Columns Transformer + Select Columns Transformer + Label␣
↪Encoder + Imputer + One Hot Encoder + Oversampler + Standard Scaler:
        Starting cross validation
```

(continues on next page)

(continued from previous page)

```
        Finished cross validation - mean Log Loss Binary: 0.617
CatBoost Classifier w/ Label Encoder + Select Columns By Type Transformer + Label␣
↪Encoder + Drop Columns Transformer + DateTime Featurizer + Imputer + Oversampler +␣
↪Select Columns Transformer + Select Columns Transformer + Label Encoder + Imputer +␣
↪Oversampler:
        Starting cross validation
        Finished cross validation - mean Log Loss Binary: 0.624
XGBoost Classifier w/ Label Encoder + Select Columns By Type Transformer + Label Encoder␣
↪+ Drop Columns Transformer + DateTime Featurizer + Imputer + Oversampler + Select␣
↪Columns Transformer + Select Columns Transformer + Label Encoder + Imputer + One Hot␣
↪Encoder + Oversampler:
        Starting cross validation
        Finished cross validation - mean Log Loss Binary: 0.271

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

Stacked Ensemble Classification Pipeline:
        Starting cross validation
        Finished cross validation - mean Log Loss Binary: 0.245

Search finished after 00:38
Best pipeline: Stacked Ensemble Classification Pipeline
Best pipeline Log Loss Binary: 0.245391
```

With the `verbose` argument set to True, the AutoML search will log its progress, reporting each pipeline and parameter set evaluated during the search.

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
```

(continues on next page)

(continued from previous page)

```
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)
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']})
```

```
Removing columns ['currency'] because they are of 'Unknown' type
Using default limit of max_batches=4.
```

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 score. The score column contains the average score across all cross-validation folds while the validation_score column is computed from the first cross-validation fold.

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

[12]:

	id	pipeline_name	search_order	mean_cv_score	standard_deviation_cv_score	validation_score	percent_better_than_baseline	high_variance_cv
0	11	Stacked Ensemble Classification Pipeline	11	0.245391	0.107989	0.245391	93.819519	False
1	10	XGBoost Classifier w/ Label Encoder + Select C...	10	0.270904	0.161665	0.270904	93.176944	False
2	4	Random Forest Classifier w/ Label Encoder + Dr...	4	0.288964	0.042651	0.288964	92.722092	False
3	2	Random Forest Classifier w/ Label Encoder + Dr...	2	0.292264	0.032252	0.292264	92.638975	False
4	6	LightGBM Classifier w/ Label Encoder + Select ...	6	0.316412	0.156072	0.316412		
5	7	Extra Trees Classifier w/ Label Encoder + Sele...	7	0.344051	0.004555	0.344051		
6	3	Logistic Regression Classifier w/ Label Encode...	3	0.573408	0.121277	0.573408		
7	8	Elastic Net Classifier w/ Label Encoder + Sele...	8	0.616802	0.195065	0.616802		
8	1	Logistic Regression Classifier w/ Label Encode...	1	0.620303	0.189890	0.620303		
9	9	CatBoost Classifier w/ Label Encoder + Select ...	9	0.623505	0.003046	0.623505		
10	5	Decision Tree Classifier w/ Label Encoder + Se...	5	3.655857	1.311022	3.655857		
11	0	Mode Baseline Binary Classification Pipeline	0	3.970423	0.266060	3.970423		

(continues on next page)

(continued from previous page)

```
4                      92.030785            False
5                      91.334658            False
6                      85.558016            False
7                      84.465076            False
8                      84.376900            False
9                      84.296249            False
10                      7.922744             True
11                      0.000000            False

                                           parameters
0   {'Label Encoder': {'positive_label': None}, 'L...
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...
6   {'Label Encoder': {'positive_label': None}, 'D...
7   {'Label Encoder': {'positive_label': None}, 'N...
8   {'Label Encoder': {'positive_label': None}, 'D...
9   {'Label Encoder': {'positive_label': None}, 'N...
10  {'Label Encoder': {'positive_label': None}, 'N...
11  {'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)
```

```
*****************************************************************************************************************
* Logistic Regression Classifier w/ Label Encoder + Drop Columns Transformer + DateTime␣
↪Featurizer + Imputer + One Hot Encoder + Oversampler + Standard Scaler *
*****************************************************************************************************************

Problem Type: binary
Model Family: Linear

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
```

(continues on next page)

(continued from previous page)

```
         * numeric_impute_strategy : mean
         * categorical_fill_value : None
         * numeric_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, 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]
         * k_neighbors : 5
7. Standard Scaler
8. Logistic Regression Classifier
         * penalty : l2
         * C : 1.0
         * n_jobs : -1
         * multi_class : auto
         * solver : lbfgs

Training
========
Training for binary problems.
Total training time (including CV): 3.9 seconds

Cross Validation
----------------
             Log Loss Binary  MCC Binary  Gini   AUC  Precision    F1  Balanced Accuracy␣
↪Binary  Accuracy Binary # Training # Validation
0                      0.440       0.101 0.284 0.642      0.250 0.167                    ␣
↪ 0.537            0.851        133           67
1                      0.602       0.246 0.144 0.572      0.400 0.308                    ␣
↪ 0.600            0.866        133           67
2                      0.819       0.070 0.031 0.516      0.136 0.207                    ␣
↪ 0.553            0.652        134           66
mean                   0.620       0.139 0.153 0.577      0.262 0.227                    ␣
↪ 0.563            0.789          -            -
std                    0.190       0.094 0.126 0.063      0.132 0.073                    ␣
↪ 0.032            0.120          -            -
coef of var            0.306       0.676 0.826 0.110      0.504 0.320                    ␣
↪ 0.058            0.151          -            -
```

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)
```

```
Logistic Regression Classifier w/ Label Encoder + Drop Columns Transformer + DateTime␣
↪Featurizer + Imputer + One Hot Encoder + Oversampler + Standard Scaler
{'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',
↪'categorical_fill_value': None, 'numeric_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, 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], 'k_neighbors': 5}, 'Logistic␣
↪Regression Classifier': {'penalty': 'l2', 'C': 1.0, 'n_jobs': -1, 'multi_class': 'auto
↪', 'solver': 'lbfgs'}}
```

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)
```

```
Stacked Ensemble Classification Pipeline
{'Label Encoder': {'positive_label': None}, 'Linear Pipeline - Label Encoder': {
↪'positive_label': None}, 'Linear Pipeline - Drop Columns Transformer': {'columns': [
↪'currency']}, 'Linear Pipeline - DateTime Featurizer': {'features_to_extract': ['year',
↪ 'month', 'day_of_week', 'hour'], 'encode_as_categories': False, 'time_index': None},
↪'Linear Pipeline - Imputer': {'categorical_impute_strategy': 'most_frequent', 'numeric_
↪impute_strategy': 'mean', 'categorical_fill_value': None, 'numeric_fill_value': None},
↪'Linear Pipeline - One Hot Encoder': {'top_n': 10, 'features_to_encode': None,
↪'categories': None, 'drop': 'if_binary', 'handle_unknown': 'ignore', 'handle_missing':
↪'error'}, 'Linear Pipeline - Oversampler': {'sampling_ratio': 0.25, 'k_neighbors_
↪default': 5, 'n_jobs': -1, 'sampling_ratio_dict': None, 'categorical_features': [3, 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], 'k_neighbors': 5}, 'Linear Pipeline - RF Classifier␣
↪Select From Model': {'number_features': None, 'n_estimators': 10, 'max_depth': None,
↪'percent_features': 0.5, 'threshold': 'median', 'n_jobs': -1}, 'Linear Pipeline -␣
↪Logistic Regression Classifier': {'penalty': 'l2', 'C': 1.0, 'n_jobs': -1, 'multi_class
↪': 'auto', 'solver': 'lbfgs'}, 'Random Forest Pipeline - Label Encoder': {'positive_
↪label': None}, 'Random Forest Pipeline - Drop Columns Transformer': {'columns': [
↪'currency']}, 'Random Forest Pipeline - DateTime Featurizer': {'features_to_extract': [
↪'year', 'month', 'day_of_week', 'hour'], 'encode_as_categories': False, 'time_index':␣
↪None}, 'Random Forest Pipeline - Imputer': {'categorical_impute_strategy': 'most_
↪frequent', 'numeric_impute_strategy': 'mean', 'categorical_fill_value': None, 'numeric_
↪fill_value': None}, 'Random Forest Pipeline - One Hot Encoder': {'top_n': 10,
↪'features_to_encode': None, 'categories': None, 'drop': 'if_binary', 'handle_unknown':
↪'ignore', 'handle_missing': 'error'}, 'Random Forest Pipeline - Oversampler': {
↪'sampling_ratio': 0.25, 'k_neighbors_default': 5, 'n_jobs': -1, 'sampling_ratio_dict':␣
↪None, 'categorical_features': [3, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26,
```

(continues on next page)

(continued from previous page)

[15]:
```
id
93     False
243    False
211    False
44      True
6      False
       ...
148    False
171    False
94     False
246    False
95     False
Name: fraud, Length: 200, 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),
 'Logistic Regression Classifier w/ Label Encoder + Drop Columns Transformer + DateTime␣
↪Featurizer + Imputer + One Hot Encoder + Oversampler + Standard Scaler': 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'], 'Standard Scaler': ['Standard␣
↪Scaler', 'Oversampler.x', 'Oversampler.y'], 'Logistic Regression Classifier': [
↪'Logistic Regression Classifier', 'Standard Scaler.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',
↪'categorical_fill_value': None, 'numeric_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, 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], 'k_neighbors': 5}, 'Logistic␣
↪Regression Classifier':{'penalty': 'l2', 'C': 1.0, 'n_jobs': -1, 'multi_class': 'auto',
↪ 'solver': 'lbfgs'}}, random_seed=0),
```

(continues on next page)

(continued from previous page)

```
 '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', 'categorical_fill_value
↪': None, 'numeric_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, 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], 'k_neighbors': 5}, 'Random Forest Classifier':{'n_estimators
↪': 100, 'max_depth': 6, 'n_jobs': -1}}, random_seed=0)}
```

[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',
               0.88),
              ('F1', 0.0),
              ('AUC', 0.5)]),
 'Logistic Regression Classifier w/ Label Encoder + Drop Columns Transformer + DateTime␣
↪Featurizer + Imputer + One Hot Encoder + Oversampler + Standard Scaler': OrderedDict([(
↪'Accuracy Binary',
               0.36),
              ('F1', 0.2),
              ('AUC', 0.5113636363636364)]),
 'Random Forest Classifier w/ Label Encoder + Drop Columns Transformer + DateTime␣
↪Featurizer + Imputer + One Hot Encoder + Oversampler': OrderedDict([('Accuracy Binary',
               0.94),
              ('F1', 0.6666666666666666),
              ('AUC', 0.8598484848484849)])}
```

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 `custom_hyperparameters` argument with your `AutoMLSearch` parameters. These parameters will limit the hyperparameter search space.

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

If however you'd like to specify certain values for the initial batch of the AutoML search algorithm, you can use the `pipeline_parameters` argument. 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
custom_hyperparameters = {'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
custom_hyperparameters = {'Imputer': {
    'numeric_impute_strategy': Categorical(['median', 'most_frequent'])
}}
# set the initial batch numeric impute strategy strategy to 'median'
pipeline_parameters = {'Imputer': {
```

(continues on next page)

(continued from previous page)

```
    'numeric_impute_strategy': 'median'
}}

# 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', and the initial␣
↪batch parameter will be
# set to 'median'
automl_constrained = AutoMLSearch(X_train=X, y_train=y, problem_type='binary',
                                  pipeline_parameters=pipeline_parameters,
                                  custom_hyperparameters=custom_hyperparameters,
                                  verbose=True)
```

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

Number of training examples: 1000
Targets
False    85.90%
True     14.10%
Name: fraud, dtype: object
Using default limit of max_batches=4.
```

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',
↪'categorical_fill_value': None, 'numeric_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}, '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)
```

(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', 'categorical_fill_value': None, 'numeric_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}
pipeline_parameters = {"Oversampler": {"sampler_balanced_ratio": sampler_ratio_dict}}
automl_auto_ratio_dict = AutoMLSearch(X_train=X, y_train=y, problem_type='binary',␣
↪pipeline_parameters=pipeline_parameters, automl_algorithm='iterative')
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}
# pipeline_parameters = {"Oversampler": {"sampler_balanced_ratio": sampler_ratio_dict}}
# automl_auto_ratio_dict = AutoMLSearch(X_train=X, y_train=y, problem_type='binary',␣
↪pipeline_parameters=pipeline_parameters)
```

(continues on next page)

(continued from previous page)

[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',
↪'categorical_fill_value': None, 'numeric_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. 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.

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

automl_with_ensembling = AutoMLSearch(X_train=X, y_train=y,
                                      problem_type="binary",
                                      allowed_model_families=[ModelFamily.LINEAR_MODEL],
                                      max_batches=4,
                                      ensembling=True,
                                      automl_algorithm='iterative',
                                      verbose=True)
automl_with_ensembling.search()
```

```
         Number of Features
Numeric                  30

Number of training examples: 569
Targets
benign       62.74%
malignant    37.26%
```

(continues on next page)

(continued from previous page)

```
Name: target, dtype: object
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
```

```
FigureWidget({
    'data': [{'mode': 'lines+markers',
              'name': 'Best Score',
              'type'...
```

```
Evaluating Baseline Pipeline: Mode Baseline Binary Classification Pipeline
Mode Baseline Binary Classification Pipeline:
        Starting cross validation
        Finished cross validation - mean Log Loss Binary: 12.868

*****************************
* 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 *
*****************************

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: 0.080
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
```

(continues on next page)

(continued from previous page)

```
        Finished cross validation - mean Log Loss Binary: 0.091
Logistic Regression Classifier w/ Label Encoder + Imputer + Standard Scaler:
        Starting cross validation
        Finished cross validation - mean Log Loss Binary: 0.097

*****************************
* 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: 0.075
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: 0.079

*****************************
* 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: 0.075387
```

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
==============
```

(continues on next page)

(continued from previous page)

```
1. Label Encoder
         * positive_label : None
2. Imputer
         * categorical_impute_strategy : most_frequent
         * numeric_impute_strategy : median
         * categorical_fill_value : None
         * numeric_fill_value : None
3. Standard Scaler
4. Elastic Net Classifier
         * penalty : elasticnet
         * C : 8.123565600467177
         * l1_ratio : 0.47997717237505744
         * 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
```

(continues on next page)

(continued from previous page)

```
# 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()
```

```
Removing columns ['currency', 'expiration_date'] because they are of 'Unknown' type
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
```

```
FigureWidget({
    'data': [{'mode': 'lines+markers',
              'name': 'Best Score',
              'type'...
```

```
Evaluating Baseline Pipeline: Mode Baseline Binary Classification Pipeline
Mode Baseline Binary Classification Pipeline:
        Starting cross validation
        Finished cross validation - mean Log Loss Binary: 4.006

*****************************
* 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: 0.577
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: 0.586
XGBoost Classifier w/ Label Encoder + Drop Columns Transformer + DateTime Featurizer +␣
↪Imputer + One Hot Encoder + Oversampler:
        Starting cross validation
```

(continues on next page)

(continued from previous page)

```
        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:
        Starting cross validation
        Finished cross validation - mean Log Loss Binary: 4.454
        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 00:18
Best pipeline: XGBoost Classifier w/ Label Encoder + Drop Columns Transformer + DateTime␣
↪Featurizer + Imputer + One Hot Encoder + Oversampler
Best pipeline Log Loss Binary: 0.266464
```

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. One can also specify `automl_algorithm='default'` using `AutoMLSearch` and it will default to using `fast` mode. 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 + Oversampler␣
↪+ 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', verbose=True)
auto_default.search()
```

```
Removing columns ['currency', 'expiration_date'] because they are of 'Unknown' type
Using default limit of max_batches=4.


*****************************
* 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:
```

```
FigureWidget({
    'data': [{'mode': 'lines+markers',
```

(continues on next page)

(continued from previous page)

```
            'name': 'Best Score',
            'type'...
Evaluating Baseline Pipeline: Mode Baseline Binary Classification Pipeline
Mode Baseline Binary Classification Pipeline:
        Starting cross validation
        Finished cross validation - mean Log Loss Binary: 4.006

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

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: 0.586
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 *
*****************************

Logistic Regression Classifier w/ Label Encoder + Drop Columns Transformer + DateTime␣
↪Featurizer + Imputer + One Hot Encoder + Oversampler + Standard Scaler + RF Classifier␣
↪Select From Model:
        Starting cross validation
        Finished cross validation - mean Log Loss Binary: 0.494
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 + Oversampler +␣
↪Select Columns Transformer + Select Columns Transformer + Label Encoder + Imputer +␣
↪One Hot Encoder + Oversampler:
        Starting cross validation
        Finished cross validation - mean Log Loss Binary: 2.267
        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 + Oversampler␣
↪+ 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 + Oversampler +␣
↪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 + Oversampler +␣
↪Select Columns Transformer + Select Columns Transformer + Label Encoder + Imputer +␣
↪One Hot Encoder + Oversampler:
        Starting cross validation
        Finished cross validation - mean Log Loss Binary: 0.350
Elastic Net Classifier w/ Label Encoder + Select Columns By Type Transformer + Label␣
↪Encoder + Drop Columns Transformer + DateTime Featurizer + Imputer + Oversampler +␣
↪Standard Scaler + Select Columns Transformer + Select Columns Transformer + Label␣
↪Encoder + Imputer + One Hot Encoder + Oversampler + Standard Scaler:
        Starting cross validation
        Finished cross validation - mean Log Loss Binary: 0.580
CatBoost Classifier w/ Label Encoder + Select Columns By Type Transformer + Label␣
↪Encoder + Drop Columns Transformer + DateTime Featurizer + Imputer + Oversampler +␣
↪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 + Oversampler + Select␣
↪Columns Transformer + Select Columns Transformer + Label Encoder + Imputer + One Hot␣
↪Encoder + Oversampler:
        Starting cross validation
        Finished cross validation - mean Log Loss Binary: 0.264

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

Stacked Ensemble Classification Pipeline:
        Starting cross validation
        Finished cross validation - mean Log Loss Binary: 0.235

Search finished after 00:34
Best pipeline: Stacked Ensemble Classification Pipeline
Best pipeline Log Loss Binary: 0.234801
```

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': 3.970423187263591,
                          'standard_deviation_cv_score': 0.26606000431837074,
                          'high_variance_cv': False,
                          'training_time': 0.2890007495880127,
                          'cv_data': [...],
                          'percent_better_than_baseline_all_objectives': {...},
                          'percent_better_than_baseline': 0,
                          'validation_score': 3.970423187263591},
                      1: {'id': 1,
                          'pipeline_name': 'Logistic Regression Classifier w/ Label Encoder␣
↪+ Drop '
                                           'Columns Transformer + DateTime Featurizer +␣
↪Imputer + One '
                                           'Hot Encoder + Oversampler + Standard Scaler',
                          'pipeline_class': <class 'evalml.pipelines.binary_classification_
↪pipeline.BinaryClassificationPipeline'>,
                          'pipeline_summary': 'Logistic Regression Classifier w/ Label␣
↪Encoder + Drop '
                                              'Columns Transformer + DateTime Featurizer +␣
↪Imputer + '
                                              'One Hot Encoder + Oversampler + Standard␣
↪Scaler',
                          'parameters': {...},
                          'mean_cv_score': 0.6203031849130022,
                          'standard_deviation_cv_score': 0.1898901497265282,
                          'high_variance_cv': False,
                          'training_time': 3.8888425827026367,
                          'cv_data': [...],
                          'percent_better_than_baseline_all_objectives': {...},
                          'percent_better_than_baseline': 84.3769000014199,
                          'validation_score': 0.6203031849130022},
                      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',
                         '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': 0.29226383693917385,
                         'standard_deviation_cv_score': 0.032252391685893105,
                         'high_variance_cv': False,
                         'training_time': 2.37630033493042,
                         'cv_data': [...],
                         'percent_better_than_baseline_all_objectives': {...},
                         'percent_better_than_baseline': 92.63897516323439,
                         'validation_score': 0.29226383693917385},
                     3: {'id': 3,
                         'pipeline_name': 'Logistic Regression Classifier w/ Label Encoder␣
↪+ Drop '
                                        'Columns Transformer + DateTime Featurizer +␣
↪Imputer + One '
                                        'Hot Encoder + Oversampler + Standard Scaler +␣
↪RF '
                                        'Classifier Select From Model',
                         'pipeline_class': <class 'evalml.pipelines.binary_classification_
↪pipeline.BinaryClassificationPipeline'>,
                         'pipeline_summary': 'Logistic Regression Classifier w/ Label␣
↪Encoder + Drop '
                                             'Columns Transformer + DateTime Featurizer +␣
↪Imputer + '
                                             'One Hot Encoder + Oversampler + Standard␣
↪Scaler + RF '
                                             'Classifier Select From Model',
                         'parameters': {...},
                         'mean_cv_score': 0.5734078967814726,
                         'standard_deviation_cv_score': 0.12127741139077003,
                         'high_variance_cv': False,
                         'training_time': 2.812499523162842,
                         'cv_data': [...],
                         'percent_better_than_baseline_all_objectives': {...},
                         'percent_better_than_baseline': 85.55801561352799,
                         'validation_score': 0.5734078967814726},
                     4: {'id': 4,
                         'pipeline_name': 'Random Forest Classifier w/ Label Encoder +␣
↪Drop Columns '
                                        'Transformer + DateTime Featurizer + Imputer +␣
↪One Hot '
                                        'Encoder + Oversampler + RF Classifier Select␣
↪From Model',
```

(continues on next page)

(continued from previous page)

```
                        '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': 0.2889637652674953,
                        'standard_deviation_cv_score': 0.042650791292308726,
                        'high_variance_cv': False,
                        'training_time': 2.722503185272217,
                        'cv_data': [...],
                        'percent_better_than_baseline_all_objectives': {...},
                        'percent_better_than_baseline': 92.72209153436239,
                        'validation_score': 0.2889637652674953},
                    5: {'id': 5,
                        'pipeline_name': 'Decision Tree Classifier w/ Label Encoder +␣
↪Select '
                                         'Columns By Type Transformer + Label Encoder +␣
↪Drop '
                                         'Columns Transformer + DateTime Featurizer +␣
↪Imputer + '
                                         'Oversampler + 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 + '
                                            'Oversampler + Select Columns Transformer +␣
↪Select '
                                            'Columns Transformer + Label Encoder +␣
↪Imputer + One '
                                            'Hot Encoder + Oversampler',
                        'parameters': {...},
                        'mean_cv_score': 3.6558567318906334,
                        'standard_deviation_cv_score': 1.311021926363768,
                        'high_variance_cv': True,
                        'training_time': 1.8442964553833008,
                        'cv_data': [...],
                        'percent_better_than_baseline_all_objectives': {...},
                        'percent_better_than_baseline': 7.922743761471843,
                        'validation_score': 3.6558567318906334},
```

(continues on next page)

(continued from previous page)

```
                  6: {'id': 6,
                      'pipeline_name': 'LightGBM Classifier w/ Label Encoder + Select␣
↪Columns By '
                                       'Type Transformer + Label Encoder + Drop Columns
↪'
                                       'Transformer + DateTime Featurizer + Imputer +␣
↪Oversampler '
                                       '+ 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␣
↪+ '
                                          'Oversampler + Select Columns Transformer +␣
↪Select '
                                          'Columns Transformer + Label Encoder +␣
↪Imputer + One '
                                          'Hot Encoder + Oversampler',
                      'parameters': {...},
                      'mean_cv_score': 0.31641155204854204,
                      'standard_deviation_cv_score': 0.15607221058209464,
                      'high_variance_cv': False,
                      'training_time': 2.061514377593994,
                      'cv_data': [...],
                      'percent_better_than_baseline_all_objectives': {...},
                      'percent_better_than_baseline': 92.03078520537726,
                      'validation_score': 0.31641155204854204},
                  7: {'id': 7,
                      'pipeline_name': 'Extra Trees Classifier w/ Label Encoder +␣
↪Select Columns '
                                       'By Type Transformer + Label Encoder + Drop␣
↪Columns '
                                       'Transformer + DateTime Featurizer + Imputer +␣
↪Oversampler '
                                       '+ Select Columns Transformer + Select Columns␣
↪Transformer '
                                       '+ Label Encoder + Imputer + One Hot Encoder +␣
↪Oversampler',
                      'pipeline_class': <class 'evalml.pipelines.binary_classification_
↪pipeline.BinaryClassificationPipeline'>,
                      'pipeline_summary': 'Extra Trees Classifier w/ Label Encoder +␣
↪Select '
                                          'Columns By Type Transformer + Label Encoder␣
↪+ Drop '
                                          'Columns Transformer + DateTime Featurizer +␣
↪Imputer + '
```

(continues on next page)

(continued from previous page)

```
                                          'Oversampler + Select Columns Transformer +␣
↪Select '
                                          'Columns Transformer + Label Encoder +␣
↪Imputer + One '
                                          'Hot Encoder + Oversampler',
                      'parameters': {...},
                      'mean_cv_score': 0.3440507438306253,
                      'standard_deviation_cv_score': 0.004555135479468251,
                      'high_variance_cv': False,
                      'training_time': 2.6381592750549316,
                      'cv_data': [...],
                      'percent_better_than_baseline_all_objectives': {...},
                      'percent_better_than_baseline': 91.33465810560752,
                      'validation_score': 0.3440507438306253},
                  8: {'id': 8,
                      'pipeline_name': 'Elastic Net Classifier w/ Label Encoder +␣
↪Select Columns '
                                       'By Type Transformer + Label Encoder + Drop␣
↪Columns '
                                       'Transformer + DateTime Featurizer + Imputer +␣
↪Oversampler '
                                       '+ Standard Scaler + Select Columns Transformer␣
↪+ Select '
                                       'Columns Transformer + Label Encoder + Imputer +␣
↪One Hot '
                                       'Encoder + Oversampler + Standard Scaler',
                      '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 + '
                                          'Oversampler + Standard Scaler + Select␣
↪Columns '
                                          'Transformer + Select Columns Transformer +␣
↪Label '
                                          'Encoder + Imputer + One Hot Encoder +␣
↪Oversampler + '
                                          'Standard Scaler',
                      'parameters': {...},
                      'mean_cv_score': 0.6168022266338659,
                      'standard_deviation_cv_score': 0.19506490263118442,
                      'high_variance_cv': False,
                      'training_time': 2.512558698654175,
                      'cv_data': [...],
                      'percent_better_than_baseline_all_objectives': {...},
                      'percent_better_than_baseline': 84.46507594927269,
                      'validation_score': 0.6168022266338659},
                  9: {'id': 9,
                      'pipeline_name': 'CatBoost Classifier w/ Label Encoder + Select␣
↪Columns By '
```

(continues on next page)

(continued from previous page)

```
                                        'Type Transformer + Label Encoder + Drop Columns
↪'
                                        'Transformer + DateTime Featurizer + Imputer +␣
↪Oversampler '
                                        '+ 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␣
↪+ '
                                             'Oversampler + Select Columns Transformer +␣
↪Select '
                                             'Columns Transformer + Label Encoder +␣
↪Imputer + '
                                             'Oversampler',
                         'parameters': {...},
                         'mean_cv_score': 0.6235053667688665,
                         'standard_deviation_cv_score': 0.003045978532889411,
                         'high_variance_cv': False,
                         'training_time': 1.8237438201904297,
                         'cv_data': [...],
                         'percent_better_than_baseline_all_objectives': {...},
                         'percent_better_than_baseline': 84.29624910591494,
                         'validation_score': 0.6235053667688665},
                     10: {'id': 10,
                          'pipeline_name': 'XGBoost Classifier w/ Label Encoder + Select␣
↪Columns By '
                                           'Type Transformer + Label Encoder + Drop␣
↪Columns '
                                           'Transformer + DateTime Featurizer + Imputer + '
                                           'Oversampler + 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 '
                                              'By Type Transformer + Label Encoder + Drop␣
↪Columns '
                                              'Transformer + DateTime Featurizer + Imputer␣
↪+ '
                                              'Oversampler + Select Columns Transformer +␣
↪Select '
                                              'Columns Transformer + Label Encoder +␣
↪Imputer + One '
```

(continues on next page)

(continued from previous page)

```
                                          'Hot Encoder + Oversampler',
                     'parameters': {...},
                     'mean_cv_score': 0.27090419801325555,
                     'standard_deviation_cv_score': 0.16166501290168173,
                     'high_variance_cv': False,
                     'training_time': 2.6334335803985596,
                     'cv_data': [...],
                     'percent_better_than_baseline_all_objectives': {...},
                     'percent_better_than_baseline': 93.17694398717326,
                     'validation_score': 0.27090419801325555},
                 11: {'id': 11,
                     'pipeline_name': 'Stacked Ensemble Classification Pipeline',
                     'pipeline_class': <class 'evalml.pipelines.binary_classification_
↪pipeline.BinaryClassificationPipeline'>,
                     'pipeline_summary': 'Stacked Ensemble Classifier w/ Label␣
↪Encoder + Label '
                                         'Encoder + Drop Columns Transformer +␣
↪DateTime '
                                         'Featurizer + Imputer + One Hot Encoder +␣
↪Oversampler '
                                         '+ Standard Scaler + RF Classifier Select␣
↪From Model + '
                                         'Logistic Regression Classifier + Label␣
↪Encoder + Drop '
                                         'Columns Transformer + DateTime Featurizer +␣
↪Imputer + '
                                         'One Hot Encoder + Oversampler + RF␣
↪Classifier Select '
                                         'From Model + Random Forest Classifier +␣
↪Label Encoder '
                                         '+ Select Columns By Type Transformer +␣
↪Label Encoder '
                                         '+ Drop Columns Transformer + DateTime␣
↪Featurizer + '
                                         'Imputer + Oversampler + Select Columns␣
↪Transformer + '
                                         'Select Columns Transformer + Label Encoder␣
↪+ Imputer '
                                         '+ One Hot Encoder + Oversampler + Decision␣
↪Tree '
                                         'Classifier + Label Encoder + Select Columns␣
↪By Type '
                                         'Transformer + Label Encoder + Drop Columns '
                                         'Transformer + DateTime Featurizer + Imputer␣
↪+ '
                                         'Oversampler + Select Columns Transformer +␣
↪Select '
                                         'Columns Transformer + Label Encoder +␣
↪Imputer + One '
                                         'Hot Encoder + Oversampler + LightGBM␣
↪Classifier + '
                                         'Label Encoder + Select Columns By Type␣
↪Transformer + '
```

(continues on next page)

(continued from previous page)

```
                                        'Label Encoder + Drop Columns Transformer +␣
↪DateTime '
                                        'Featurizer + Imputer + Oversampler + Select␣
↪Columns '
                                        'Transformer + Select Columns Transformer +␣
↪Label '
                                        'Encoder + Imputer + One Hot Encoder +␣
↪Oversampler + '
                                        'Extra Trees Classifier + Label Encoder +␣
↪Select '
                                        'Columns By Type Transformer + Label Encoder␣
↪+ Drop '
                                        'Columns Transformer + DateTime Featurizer +␣
↪Imputer + '
                                        'Oversampler + Select Columns Transformer +␣
↪Select '
                                        'Columns Transformer + Label Encoder +␣
↪Imputer + '
                                        'Oversampler + CatBoost Classifier + Label␣
↪Encoder + '
                                        'Select Columns By Type Transformer + Label␣
↪Encoder + '
                                        'Drop Columns Transformer + DateTime␣
↪Featurizer + '
                                        'Imputer + Oversampler + Select Columns␣
↪Transformer + '
                                        'Select Columns Transformer + Label Encoder␣
↪+ Imputer '
                                        '+ One Hot Encoder + Oversampler + XGBoost␣
↪Classifier',
                      'parameters': {...},
                      'mean_cv_score': 0.24539126694570923,
                      'standard_deviation_cv_score': 0.10798917363618948,
                      'high_variance_cv': False,
                      'training_time': 9.815217733383179,
                      'cv_data': [...],
                      'percent_better_than_baseline_all_objectives': {...},
                      'percent_better_than_baseline': 93.81951859104389,
                      'validation_score': 0.24539126694570923,
                      'input_pipeline_ids': [...]}},
'search_order': [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11]}
```

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.

[33]:
```
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(show_iteration_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 + Oversampler␣
↪+ 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.

[34]:
```
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(show_iteration_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 + Oversampler␣
↪+ Select Columns Transformer + Select Columns Transformer + Label Encoder + Imputer +␣
↪One Hot Encoder + Oversampler may not perform as estimated on unseen data.
```

(continues on next page)

(continued from previous page)

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(show_iteration_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.

[35]:
```
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(show_iteration_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 + Oversampler␣
↪+ Select Columns Transformer + Select Columns Transformer + Label Encoder + Imputer +␣
↪One Hot Encoder + Oversampler may not perform as estimated on unseen data.
/home/docs/checkouts/readthedocs.org/user_builds/feature-labs-inc-evalml/envs/v0.48.0/
↪lib/python3.8/site-packages/distributed/worker.py:4278: UserWarning:

Large object of size 2.57 MiB detected in task graph:
  {'pipeline': pipeline = BinaryClassificationPipeli ... x7fb83bd94460>}
Consider scattering large objects ahead of time
with client.scatter to reduce scheduler burden and
keep data on workers

    future = client.submit(func, big_data)    # bad

    big_future = client.scatter(big_data)     # good
```

(continues on next page)

(continued from previous page)

```
    future = client.submit(func, big_future)  # good

Loky-backed parallel loops cannot be called in a multiprocessing, setting n_jobs=1
Loky-backed parallel loops cannot be called in a multiprocessing, setting n_jobs=1
Loky-backed parallel loops cannot be called in a multiprocessing, setting n_jobs=1
Loky-backed parallel loops cannot be called in a multiprocessing, setting n_jobs=1
Loky-backed parallel loops cannot be called in a multiprocessing, setting n_jobs=1
Loky-backed parallel loops cannot be called in a multiprocessing, setting n_jobs=1
Loky-backed parallel loops cannot be called in a multiprocessing, setting n_jobs=1
Loky-backed parallel loops cannot be called in a multiprocessing, setting n_jobs=1
Loky-backed parallel loops cannot be called in a multiprocessing, setting n_jobs=1
Loky-backed parallel loops cannot be called in a multiprocessing, setting n_jobs=1
Loky-backed parallel loops cannot be called in a multiprocessing, setting n_jobs=1
Loky-backed parallel loops cannot be called in a multiprocessing, setting n_jobs=1
Loky-backed parallel loops cannot be called in a multiprocessing, setting n_jobs=1
Loky-backed parallel loops cannot be called in a multiprocessing, setting n_jobs=1
Loky-backed parallel loops cannot be called in a multiprocessing, setting n_jobs=1
Loky-backed parallel loops cannot be called in a multiprocessing, setting n_jobs=1
Loky-backed parallel loops cannot be called in a multiprocessing, setting n_jobs=1
Loky-backed parallel loops cannot be called in a multiprocessing, setting n_jobs=1
Loky-backed parallel loops cannot be called in a multiprocessing, setting n_jobs=1
Loky-backed parallel loops cannot be called in a multiprocessing, setting n_jobs=1
Loky-backed parallel loops cannot be called in a multiprocessing, setting n_jobs=1
Loky-backed parallel loops cannot be called in a multiprocessing, setting n_jobs=1
Loky-backed parallel loops cannot be called in a multiprocessing, setting n_jobs=1
Loky-backed parallel loops cannot be called in a multiprocessing, setting n_jobs=1
Loky-backed parallel loops cannot be called in a multiprocessing, setting n_jobs=1
Loky-backed parallel loops cannot be called in a multiprocessing, setting n_jobs=1
Loky-backed parallel loops cannot be called in a multiprocessing, setting n_jobs=1
Loky-backed parallel loops cannot be called in a multiprocessing, setting n_jobs=1
Loky-backed parallel loops cannot be called in a multiprocessing, setting n_jobs=1
Loky-backed parallel loops cannot be called in a multiprocessing, setting n_jobs=1
Loky-backed parallel loops cannot be called in a multiprocessing, setting n_jobs=1
Loky-backed parallel loops cannot be called in a multiprocessing, setting n_jobs=1
Loky-backed parallel loops cannot be called in a multiprocessing, setting n_jobs=1
Loky-backed parallel loops cannot be called in a multiprocessing, setting n_jobs=1
Loky-backed parallel loops cannot be called in a multiprocessing, setting n_jobs=1
Loky-backed parallel loops cannot be called in a multiprocessing, setting n_jobs=1
Loky-backed parallel loops cannot be called in a multiprocessing, setting n_jobs=1
Loky-backed parallel loops cannot be called in a multiprocessing, setting n_jobs=1
Loky-backed parallel loops cannot be called in a multiprocessing, setting n_jobs=1
Loky-backed parallel loops cannot be called in a multiprocessing, setting n_jobs=1
Loky-backed parallel loops cannot be called in a multiprocessing, setting n_jobs=1
Loky-backed parallel loops cannot be called in a multiprocessing, setting n_jobs=1
Loky-backed parallel loops cannot be called in a multiprocessing, setting n_jobs=1
Loky-backed parallel loops cannot be called in a multiprocessing, setting n_jobs=1
Loky-backed parallel loops cannot be called in a multiprocessing, setting n_jobs=1
Loky-backed parallel loops cannot be called in a multiprocessing, setting n_jobs=1
Loky-backed parallel loops cannot be called in a multiprocessing, setting n_jobs=1
```

[36]:

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

(continues on next page)

(continued from previous page)

```
automl_dask_t4 = AutoMLSearch(X_train=X, y_train=y,
                              problem_type="binary",
                              allowed_model_families=[ModelFamily.LINEAR_MODEL],
                              engine=dask_engine_t4)
automl_dask_t4.search(show_iteration_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 + Oversampler␣
↪+ 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!

[37]:
```
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: 38.029608488082886
Concurrent futures (threaded) search duration: 39.36334729194641
Dask (two processes) search duration: 43.02927899360657
Dask (four threads)search duration: 45.0988609790802
```

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', 'categorical_fill_value': None, 'numeric_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', 'categorical_fill_value
↪': None, 'numeric_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', 'categorical_fill_value
↪': None, 'numeric_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(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.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', 0.04132737017536148)])
```

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)


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',
↪'categorical_fill_value': None, 'numeric_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'],
```

(continues on next page)

(continued from previous page)

```
    '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
         * categorical_fill_value : None
         * numeric_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 *
*****************************************************************************************************************
```

(continues on next page)

(continued from previous page)

```
Problem Type: multiclass
Model Family: Linear

Pipeline Steps
==============
1. Imputer
         * categorical_impute_strategy : most_frequent
         * numeric_impute_strategy : mean
         * categorical_fill_value : None
         * numeric_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
         * 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',␣
      ↪categorical_fill_value=None, numeric_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',
```

(continues on next page)

(continued from previous page)

```
 '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()
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
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
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
```

(continues on next page)

(continued from previous page)

```
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

     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', 'categorical_fill_value': None, 'numeric_fill_value': None},
↪'Random Forest Classifier':{'n_estimators': 100, 'max_depth': 6, 'n_jobs': -1}},␣
↪random_seed=0)
```

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.
```

(continues on next page)

(continued from previous page)

```
                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)
        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)
```

(continues on next page)

(continued from previous page)

```
# 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
↪'], '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', 'categorical_fill_value': None, 'numeric_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

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', 'categorical_fill_value': None, 'numeric_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', categorical_fill_value=None, numeric_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]:
```
['OHE', 'My Imputer', '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",
        ],
}
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
         * categorical_fill_value : None
         * numeric_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
```

(continues on next page)

(continued from previous page)

```
        * 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
```

(continues on next page)

(continued from previous page)

```
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
..                ...             ...                  ...            ...
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
```

(continues on next page)

(continued from previous page)

```
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
..                       ...
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: Drop NaN Rows Transformer
Transformer: Replace Nullable Types Transformer
Transformer: Drop Rows Transformer
Transformer: URL Featurizer
Transformer: Email Featurizer
Transformer: Log Transformer
Transformer: Polynomial Detrender
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
Transformer: Target Imputer
Transformer: Imputer
Transformer: Per Column Imputer
Transformer: Simple Imputer
Transformer: RF Regressor Select From Model
Transformer: RF Classifier Select From Model
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
```

(continues on next page)

(continued from previous page)

```
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
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.

        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:
```

(continues on next page)

(continued from previous page)

```
            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
        """
        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 = {}
    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":
```

(continues on next page)

(continued from previous page)

```
            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:
            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.

4.5.2 Core Objectives

Use the `get_core_objectives` method to get a list of which objectives are included with EvalML for each problem type:

[1]:
```
from evalml.objectives import get_core_objectives
from evalml.problem_types import ProblemTypes

for objective in get_core_objectives(ProblemTypes.BINARY):
    print(objective.name)
```

```
MCC Binary
Log Loss Binary
Gini
AUC
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.

[2]:
```
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=['Simple Imputer', 'DateTime␣
↪Featurizer', 'One Hot Encoder', 'Random Forest Classifier'])
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)])
```

4.5.3 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.

[3]:
```
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=.5, interchange_fee=.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.
```

(continues on next page)

(continued from previous page)

```
                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

    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)
```

(continues on next page)

(continued from previous page)

```
        # 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
        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 Graphing Utilities

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)
```

(continues on next page)

(continued from previous page)

```
pipeline_binary = BinaryClassificationPipeline(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_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
```

(continues on next page)

(continued from previous page)

```
21  fractal dimension error    0.005502
22        compactness error    0.004953
23         smoothness error    0.004728
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

Permutation Importance

We can also compute and plot the permutation importance of the pipeline.

[4]:
```
from evalml.model_understanding import calculate_permutation_importance
calculate_permutation_importance(pipeline_binary, X_holdout, y_holdout, 'log loss binary
↪')
```

[4]:
```
                    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
```

(continues on next page)

(continued from previous page)

```
25  worst fractal dimension   -0.007415
26           symmetry error   -0.008175
27         perimeter error    -0.008980
28    concave points error    -0.010415
29           worst symmetry   -0.018645
```

[5]:
```
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.

[6]:
```
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.
```

[7]:
```
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.

Partial Dependence Plots

We can calculate the one-way partial dependence plots for a feature.

[8]:
```
from evalml.model_understanding import partial_dependence
partial_dependence(pipeline_binary, X_holdout, features='mean radius', grid_resolution=5)
```

[8]:
```
   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
```

[9]:
```
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

You can also compute the partial dependence for a categorical feature. We will demonstrate this on the fraud dataset.

[10]:
```
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.

[11]:
```
partial_dependence(pipeline_binary, X_holdout, features=('worst perimeter', 'worst radius
↪'), grid_resolution=5)
```

[11]:
```
             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
```

(continues on next page)

(continued from previous page)

```
144.721425   malignant
169.915000   malignant
```

[12]:
```
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

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.

[13]:
```
from evalml.model_understanding.graphs import confusion_matrix
y_pred = pipeline_binary.predict(X_holdout)
confusion_matrix(y_holdout, y_pred)
```

[13]:
```
              benign  malignant
benign      0.930556   0.069444
malignant   0.023810   0.976190
```

[14]:
```
from evalml.model_understanding.graphs 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.

[15]:
```
from evalml.model_understanding.graphs 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.

[16]:
```
from evalml.model_understanding.graphs 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.

[17]:
```
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

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`.

[18]:
```
from evalml.model_understanding.graphs import binary_objective_vs_threshold
binary_objective_vs_threshold(pipeline_binary, X_holdout, y_holdout, 'f1', steps=10)
```

[18]:

	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

(continues on next page)

(continued from previous page)

```
5          0.5  0.931818
6          0.6  0.941176
7          0.7  0.951220
8          0.8  0.936709
9          0.9  0.923077
10         1.0  0.000000
```

[19]:
```
from evalml.model_understanding.graphs 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.

[20]:
```
from evalml.model_understanding.graphs 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
200.0    1.36%
72.0     1.36%
90.0     1.13%
178.0    1.13%
71.0     1.13%
         ...
73.0     0.23%
222.0    0.23%
86.0     0.23%
79.0     0.23%
57.0     0.23%
Name: target, Length: 214, dtype: object
```

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

Now let's train a decision tree on some data.

[21]:
```
pipeline_dt = BinaryClassificationPipeline(['Simple Imputer', 'Decision Tree Classifier
↪'])
pipeline_dt.fit(X_train, y_train)
```

[21]:
```
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)
```

Tree Visualization

We can visualize the structure of the Decision Tree that was fit to that data, and save it if necessary.

[22]:
```
from evalml.model_understanding.graphs import visualize_decision_tree

visualize_decision_tree(pipeline_dt.estimator, max_depth=2, rotate=False, filled=True,␣
↪filepath=None)
```

[22]:

4.6.2 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%.

[23]:
```
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)
print(table)
```

```
Random Forest Classifier w/ Label Encoder + Imputer

{'Label Encoder': {'positive_label': None}, 'Imputer': {'categorical_impute_strategy':
↪'most_frequent', 'numeric_impute_strategy': 'mean', 'categorical_fill_value': None,
↪'numeric_fill_value': None}, 'Random Forest Classifier': {'n_estimators': 100, 'max_
↪depth': 6, 'n_jobs': -1}}
```

(continues on next page)

(continued from previous page)

```
        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*.

[24]:
```
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

{'Label Encoder': {'positive_label': None}, 'Imputer': {'categorical_impute_strategy':
↪'most_frequent', 'numeric_impute_strategy': 'mean', 'categorical_fill_value': None,
↪'numeric_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
```

(continues on next page)

(continued from previous page)

```
                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.

[25]:
```
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', 'categorical_fill_value': None,
↪'numeric_fill_value': None}, 'Random Forest Classifier': {'n_estimators': 100, 'max_
↪depth': 6, 'n_jobs': -1}}


        1 of 1
```

(continues on next page)

(continued from previous page)

```
                   Feature Name       Feature Value   Contribution to Prediction   LIME␣
↪Value
              ␣
↪==============================================================================
                 worst perimeter          92.80                  +                   0.
↪06
                   worst radius           14.04                  +                   0.
↪06
                    worst area           599.50                  +                   0.
↪05
               mean concave points        0.03                   +                   0.
↪04
               worst concave points       0.12                   +                   0.
↪04
                 worst concavity          0.18                   +                   0.
↪03
```

[26]:
```
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")
print(report)
```

```
Random Forest Classifier w/ Label Encoder + Imputer

{'Label Encoder': {'positive_label': None}, 'Imputer': {'categorical_impute_strategy':
↪'most_frequent', 'numeric_impute_strategy': 'mean', 'categorical_fill_value': None,
↪'numeric_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 radius         11.94                  +                    0.06
                worst perimeter       80.78                  +                    0.06
                  worst area         433.10                  +                    0.05
```

(continues on next page)

(continued from previous page)

```
        3 of 3

                 Feature Name     Feature Value   Contribution to Prediction   LIME Value
                ==============================================================================
                worst perimeter       99.21                   +                    0.06
                 worst radius         14.42                   +                    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.

[27]:
```
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', 'categorical_fill_value': None,
↪'numeric_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
```

(continues on next page)

(continued from previous page)

```
                 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
               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
```

(continues on next page)

(continued from previous page)

```
                 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
               ␣
↪================================================================================
                   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
```

[28]:
```
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', 'categorical_fill_value': None,
↪'numeric_fill_value': None}, 'Random Forest Classifier': {'n_estimators': 100, 'max_
↪depth': 6, 'n_jobs': -1}}
```

(continues on next page)

(continued from previous page)

```
       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 radius           13.57                    +                  0.
↪06
                   worst perimeter          86.67                    +                  0.
↪06
                      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.
↪02


       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 perimeter          83.67                    +                  0.
↪06
                     worst radius           13.10                    +                  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.
↪02
```

(continues on next page)

(continued from previous page)

```
       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
               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
```

(continues on next page)

(continued from previous page)

We use a custom metric (hinge loss) for selecting the best and worst predictions. See this example:

[29]:
```
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)

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', 'categorical_fill_value': None,
↪'numeric_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
```

(continues on next page)

(continued from previous page)

```
                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
                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
```

(continues on next page)

(continued from previous page)

```
                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
                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
```

(continues on next page)

(continued from previous page)

```
                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


        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
```

(continues on next page)

(continued from previous page)

```
                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
                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

[30]:
```
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,
```

(continues on next page)

(continued from previous page)


```
            14.04,
            0.034,
            92.8
          ],
          "qualitative_explanation": [
            "-",
            "-",
            "-",
            "-",
            "-",
            "-"
          ],
          "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

[31]:
```
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
```

[31]:
```
         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
```

(continues on next page)

(continued from previous page)

```
3                 -0.046667  malignant                 0
4                 -0.054151  malignant                 0
5                 -0.055237  malignant                 0
```

Best and worst predictions as a dictionary

[32]:
```
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": 0.00019704435070700 75,
        "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,
```


(continues on next page)

(continued from previous page)


```
            0.02704
          ],
          "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

[33]:
```
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
```

[33]:
```
           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                       -

    quantitative_explanation class_name  label_benign_probability  \
```

(continues on next page)

(continued from previous page)

```
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
```

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.

[34]:
```
import shap

from evalml.model_understanding.force_plots import graph_force_plot
```

(continues on next page)

(continued from previous page)

```
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 0x7fb8f1ac9610>
```

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.

[35]:
```
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 0x7fb8f15c3400>
```

```
Class: class_1
```

```
<shap.plots._force.AdditiveForceVisualizer at 0x7fb8f221e820>
```

```
Class: class_2
```

```
<shap.plots._force.AdditiveForceVisualizer at 0x7fb8f0513ca0>
```

```
Row: 1
Class: class_0
```

```
<shap.plots._force.AdditiveForceVisualizer at 0x7fb8f0513760>
```

```
Class: class_1
```

```
<shap.plots._force.AdditiveForceVisualizer at 0x7fb8f0513ee0>
```

```
Class: class_2
```

```
<shap.plots._force.AdditiveForceVisualizer at 0x7fb8f0513cd0>
Row: 2
Class: class_0
<shap.plots._force.AdditiveForceVisualizer at 0x7fb8f0513be0>
Class: class_1
<shap.plots._force.AdditiveForceVisualizer at 0x7fb8f0513b50>
Class: class_2
<shap.plots._force.AdditiveForceVisualizer at 0x7fb8f0513370>
Row: 3
Class: class_0
<shap.plots._force.AdditiveForceVisualizer at 0x7fb8f0513190>
Class: class_1
<shap.plots._force.AdditiveForceVisualizer at 0x7fb8f0513dc0>
Class: class_2
<shap.plots._force.AdditiveForceVisualizer at 0x7fb8f0513a90>
Row: 4
Class: class_0
<shap.plots._force.AdditiveForceVisualizer at 0x7fb8f05138e0>
Class: class_1
<shap.plots._force.AdditiveForceVisualizer at 0x7fb8f0513e20>
Class: class_2
<shap.plots._force.AdditiveForceVisualizer at 0x7fb8f0513af0>
```

4.6.3 Find 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.

[36]:
```
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)
```

[36]:

	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]
```

[37]:
```
objective_thresholds
```

[37]:
```
{'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.

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)

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: '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)

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: 'no var col with nan' has two unique values including nulls. Consider encoding␣
↪the nulls for this column to be useful for machine learning.
Error: 'no var col' has 1 unique value.
```

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({'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])
```

(continues on next page)

(continued from previous page)

```
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'])
```

```
Warning: Columns 'leak', 'x', 'y' are 80.0% or more correlated with the target
```

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 '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', '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 'id', 'user_number' 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.

[8]:
```
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'])
```

```
Warning: Columns are likely to be correlated: [('col_1', 'col_2'), ('col_1', 'col_3'), (
↪'col_1', 'col_4'), ('col_1', 'col_5'), ('col_2', 'col_3'), ('col_2', 'col_4'), ('col_2
↪', 'col_5'), ('col_3', 'col_4'), ('col_3', 'col_5'), ('col_4', 'col_5')]
```

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.

[9]:
```
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=.5)
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.

[10]:
```
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=.4,
                                   unique_count_threshold=3)
messages = sparsity_check.validate(X)
```

(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: 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.

[11]:
```
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.

[12]:
```
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.

[13]:
```
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)

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).

[14]:
```
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.
```

(continues on next page)

(continued from previous page)

```
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("--------------------------------")

# 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 missing between start and end date.
--------------------------------
Error: No frequency could be detected in column 'dates', possibly due to uneven␣
↪intervals.
Error: Datetime values must be sorted in ascending order.
```

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.

[15]:
```
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
↪"}
```

(continues on next page)

(continued from previous page)

```
# 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']
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: Since the data has 100 observations and n_splits=3, the smallest split would have␣
↪25 observations. Since 25 (gap + max_delay + forecast_horizon) >= 25, 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.
```

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.

[16]:
```
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:
```

(continues on next page)

(continued from previous page)

```
    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.

[17]:
```
from evalml.data_checks import NoVarianceDataCheck, DataCheckError, 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)

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: 'no var col with nan' has two unique values including nulls. Consider encoding␣
↪the nulls for this column to be useful for machine learning.
Error: 'no var col' has 1 unique value.
```

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.

[18]:
```
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)
```

(continues on next page)

(continued from previous page)

```
        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])
        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.

[19]:
```
# 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 = [NullDataCheck, InvalidTargetDataCheck, NoVarianceDataCheck,␣
↪TargetLeakageDataCheck]

    def __init__(self, problem_type, objective):
        """
```

(continues on next page)

(continued from previous page)

```
        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
```

[20]:
```
# Pass list of data checks to the `data_checks` parameter of DataChecks
same_custom_data_checks = DataChecks(data_checks=[NullDataCheck, InvalidTargetDataCheck,␣
↪NoVarianceDataCheck, 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

Number of training examples: 1500
Targets
False    86.60%
True     13.40%
Name: fraud, dtype: object
```

[2]:

id	card_id	store_id	datetime	amount	currency	customer_present	expiration_date	provider	lat	lng	region
0	32261	8516	2019-01-01 00:12:26	24900	CUC	True	08/24	Mastercard	38.58894	-89.99038	Fairview Heights
1	16434	8516	2019-01-01 09:42:03	15789	MYR	False	11/21	Discover	38.58894	-89.99038	Fairview Heights
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					

(continues on next page)

(continued from previous page)

```
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
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]:
```
# let's copy the datetime at row 1 for future use
date = X_train.iloc[1]['datetime']

# 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)
# Let's take another look at the new X_train data
X_train
```

[4]:
```
      card_id  store_id            datetime   amount currency  \
id
872   15492.0    2868.0 2019-08-03 02:50:04  80719.0      HNL
1477      NaN       NaN                 NaT      NaN      NaN
158   22440.0    6813.0 2019-07-12 11:07:25   1849.0      SEK
808    8096.0    8096.0 2019-06-11 21:33:36  41358.0      MOP
336   33270.0    1529.0 2019-03-23 21:44:00  32594.0      CUC
...       ...       ...                 ...      ...      ...
339    8484.0    5358.0 2019-01-10 07:47:28  89503.0      GMD
1383  17565.0    3929.0 2019-01-15 01:11:02  14264.0      DKK
893     108.0      44.0 2019-05-17 00:53:39  93218.0      SLL
385   29983.0     152.0 2019-06-09 06:50:29  41105.0      RWF
1074  26197.0    4927.0 2019-05-22 15:57:27  50481.0      MNT

     customer_present expiration_date          provider       lat        lng  \
id
872              True           08/27  American Express   5.47090  100.24529
1477              NaN             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          1.0           NaN
1477            NaN     NaN          NaN           NaN
158            Jais      IN          1.0           NaN
808           Narva      EE          1.0           NaN
336          Strood      GB          1.0           NaN
...             ...     ...          ...           ...
339        Adliswil      CH          1.0           NaN
1383     Rudolstadt      DE          1.0           NaN
893          Burgos      PH          1.0           NaN
385        Magomeni      TZ          1.0           NaN
1074      Scarsdale      US          1.0           NaN

[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")
```

(continues on next page)

(continued from previous page)

```
    print("Search errored out! Message received is: {}".format(e))
    print("=" * 80, "\n")
```

```
================================================================================

Search errored out! Message received is: Input contains NaN, infinity or a value too␣
↪large for dtype('float64').
================================================================================
```

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'],
    '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': "Column(s) 'card_id', 'store_id', 'amount', 'currency', 'customer_present',
↪ 'expiration_date', 'provider', 'lat', 'lng', 'region', 'country', 'no_variance' have␣
↪null values",
   'data_check_name': 'NullDataCheck',
   'level': 'warning',
   'details': {'columns': ['card_id',
     'store_id',
     'amount',
     'currency',
```

(continues on next page)

(continued from previous page)

```
    'customer_present',
    'expiration_date',
    'provider',
    'lat',
    'lng',
    'region',
    'country',
    'no_variance'],
   'rows': None},
  'code': 'COLS_WITH_NULL',
  'action_options': [{'code': 'IMPUTE_COL',
    'data_check_name': 'NullDataCheck',
    'metadata': {'columns': ['card_id',
      'store_id',
      'amount',
      'currency',
      'customer_present',
      'expiration_date',
      'provider',
      'lat',
      'lng',
      'region',
      'country',
      'no_variance'],
     'rows': None,
     'is_target': False},
    'parameters': {'impute_strategies': {'parameter_type': 'column',
      'columns': {'card_id': {'impute_strategy': {'categories': ['mean',
          'most_frequent'],
        'type': 'category',
        'default_value': 'mean'}},
       'store_id': {'impute_strategy': {'categories': ['mean',
          'most_frequent'],
        'type': 'category',
        'default_value': 'mean'}},
       'amount': {'impute_strategy': {'categories': ['mean', 'most_frequent'],
        'type': 'category',
        'default_value': 'mean'}},
       'currency': {'impute_strategy': {'categories': ['most_frequent'],
        'type': 'category',
        'default_value': 'most_frequent'}},
       'customer_present': {'impute_strategy': {'categories': ['most_frequent'],
        'type': 'category',
        'default_value': 'most_frequent'}},
       'expiration_date': {'impute_strategy': {'categories': ['most_frequent'],
        'type': 'category',
        'default_value': 'most_frequent'}},
       'provider': {'impute_strategy': {'categories': ['most_frequent'],
        'type': 'category',
        'default_value': 'most_frequent'}},
       'lat': {'impute_strategy': {'categories': ['mean', 'most_frequent'],
        'type': 'category',
```

(continues on next page)

(continued from previous page)

```
          'default_value': 'mean'}},
        'lng': {'impute_strategy': {'categories': ['mean', 'most_frequent'],
          'type': 'category',
          'default_value': 'mean'}},
        'region': {'impute_strategy': {'categories': ['most_frequent'],
          'type': 'category',
          'default_value': 'most_frequent'}},
        'country': {'impute_strategy': {'categories': ['most_frequent'],
          'type': 'category',
          'default_value': 'most_frequent'}},
        'no_variance': {'impute_strategy': {'categories': ['mean',
           'most_frequent'],
          'type': 'category',
          'default_value': 'mean'}}}}}}]},
 {'message': '1 row(s) (0.08333333333333334%) of target values are null',
  'data_check_name': 'InvalidTargetDataCheck',
  'level': 'error',
  'details': {'columns': None,
   'rows': None,
   'num_null_rows': 1,
   'pct_null_rows': 0.08333333333333334},
  'code': 'TARGET_HAS_NULL',
  'action_options': [{'code': 'IMPUTE_COL',
    'data_check_name': 'InvalidTargetDataCheck',
    'metadata': {'columns': None, 'rows': None, 'is_target': True},
    'parameters': {'impute_strategy': {'parameter_type': 'global',
      'type': 'category',
      'categories': ['most_frequent'],
      'default_value': 'most_frequent'}}}]},
 {'message': "'no_variance' has 1 unique value.",
  'data_check_name': 'NoVarianceDataCheck',
  'level': 'error',
  '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)))
```

```
The new length of X_train is 1199 and y_train is 1199
```

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   2  Random Forest Classifier w/ Label Encoder + Da...             2
1   1  Logistic Regression Classifier w/ Label Encode...             1
2   0       Mode Baseline Binary Classification Pipeline             0

   mean_cv_score  standard_deviation_cv_score  validation_score  \
0       0.269118                     0.007188          0.269118
1       0.393342                     0.022323          0.393342
2       4.637776                     0.043230          4.637776

   percent_better_than_baseline  high_variance_cv  \
0                     94.197251             False
1                     91.518737             False
2                      0.000000             False

                                      parameters
0  {'Label Encoder': {'positive_label': None}, 'D...
1  {'Label Encoder': {'positive_label': None}, 'D...
2  {'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',
  'level': 'error',
  'details': {'columns': None,
   'rows': None,
   'num_null_rows': 1,
   'pct_null_rows': 0.08333333333333334},
  'code': 'TARGET_HAS_NULL',
  'action_options': [{'code': 'IMPUTE_COL',
    'data_check_name': 'InvalidTargetDataCheck',
    'metadata': {'columns': None, 'rows': None, 'is_target': True},
    'parameters': {'impute_strategy': {'parameter_type': 'global',
      'type': 'category',
      'categories': ['most_frequent'],
      'default_value': 'most_frequent'}}}]},
 {'message': "'no_variance' has 1 unique value.",
  'data_check_name': 'NoVarianceDataCheck',
  'level': 'error',
  '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': {}}]}]
```

[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

# first, let's address the `TARGET_HAS_NULL` error
y_train_no_errors.fillna(False, inplace=True)

# here, we address the `NO_VARIANCE` error
X_train_no_errors.drop("no_variance", axis=1, inplace=True)

# lastly, we address the `DATETIME_HAS_NAN` error with the date we had saved earlier
X_train_no_errors.iloc[1, 2] = date

# let's reinitialize the Woodwork DataTable
X_train_no_errors.ww.init()
```

(continues on next page)

(continued from previous page)

```
X_train_no_errors.head()
```

[12]:
```
      card_id  store_id            datetime   amount currency  \
id
872   15492.0    2868.0 2019-08-03 02:50:04  80719.0      HNL
1477      NaN       NaN 2019-08-05 21:05:57      NaN      NaN
158   22440.0    6813.0 2019-07-12 11:07:25   1849.0      SEK
808    8096.0    8096.0 2019-06-11 21:33:36  41358.0      MOP
336   33270.0    1529.0 2019-03-23 21:44:00  32594.0      CUC

     customer_present expiration_date          provider       lat        lng  \
id
872              True           08/27  American Express   5.47090  100.24529
1477              NaN             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  mostly_nulls
id
872   Batu Feringgi      MY           NaN
1477            NaN     NaN           NaN
158            Jais      IN           NaN
808           Narva      EE           NaN
336          Strood      GB           NaN
```

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 0x7f841cbff550>,
 [{'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': "Column(s) 'card_id', 'store_id', 'amount', 'currency', 'customer_present',
↪ 'expiration_date', 'provider', 'lat', 'lng', 'region', 'country' have null values",
   'data_check_name': 'NullDataCheck',
   'level': 'warning',
   'details': {'columns': ['card_id',
     'store_id',
     'amount',
```

(continues on next page)

(continued from previous page)

```
    'currency',
    'customer_present',
    'expiration_date',
    'provider',
    'lat',
    'lng',
    'region',
    'country'],
   'rows': None},
  'code': 'COLS_WITH_NULL',
  'action_options': [{'code': 'IMPUTE_COL',
    'data_check_name': 'NullDataCheck',
    'metadata': {'columns': ['card_id',
      'store_id',
      'amount',
      'currency',
      'customer_present',
      'expiration_date',
      'provider',
      'lat',
      'lng',
      'region',
      'country'],
     'rows': None,
     'is_target': False},
    'parameters': {'impute_strategies': {'parameter_type': 'column',
      'columns': {'card_id': {'impute_strategy': {'categories': ['mean',
          'most_frequent'],
        'type': 'category',
        'default_value': 'mean'}},
       'store_id': {'impute_strategy': {'categories': ['mean',
          'most_frequent'],
        'type': 'category',
        'default_value': 'mean'}},
       'amount': {'impute_strategy': {'categories': ['mean', 'most_frequent'],
        'type': 'category',
        'default_value': 'mean'}},
       'currency': {'impute_strategy': {'categories': ['most_frequent'],
        'type': 'category',
        'default_value': 'most_frequent'}},
       'customer_present': {'impute_strategy': {'categories': ['most_frequent'],
        'type': 'category',
        'default_value': 'most_frequent'}},
       'expiration_date': {'impute_strategy': {'categories': ['most_frequent'],
        'type': 'category',
        'default_value': 'most_frequent'}},
       'provider': {'impute_strategy': {'categories': ['most_frequent'],
        'type': 'category',
        'default_value': 'most_frequent'}},
       'lat': {'impute_strategy': {'categories': ['mean', 'most_frequent'],
        'type': 'category',
        'default_value': 'mean'}},
```

(continues on next page)

(continued from previous page)

```
        'lng': {'impute_strategy': {'categories': ['mean', 'most_frequent'],
          'type': 'category',
          'default_value': 'mean'}},
        'region': {'impute_strategy': {'categories': ['most_frequent'],
          'type': 'category',
          'default_value': 'most_frequent'}},
        'country': {'impute_strategy': {'categories': ['most_frequent'],
          'type': 'category',
          'default_value': 'most_frequent'}}}}}}]}])
```

4.8.4 Comparing removing only errors versus removing both warnings and errors

Let's see the differences in model performance when we remove only errors versus remove both warnings and errors. To do this, we compare the performance of the best pipelines on the validation data. Remember that in the search where we only address errors, we still have the `mostly_nulls` column present in the data, so we leave that column in the validation data for its respective search. We drop the other `no_variance` column from both searches.

Additionally, we do some logical type setting since we had added additional noise to just the training data. This allows the data to be of the same types in both training and validation.

[14]:
```
# drop the no_variance column
X_valid.drop("no_variance", axis=1, inplace=True)

# logical type management
X_valid.ww.init(logical_types={"customer_present": "Categorical"})
y_valid = ww.init_series(y_valid, logical_type="Categorical")

best_pipeline_no_errors = results_no_errors[0].best_pipeline
print("Only dropping errors:", best_pipeline_no_errors.score(X_valid, y_valid, ["Log␣
↪Loss Binary"]), "\n")

# drop the mostly_nulls column and reinitialize the DataTable
X_valid.drop("mostly_nulls", axis=1, inplace=True)
X_valid.ww.init()

best_pipeline_clean = results_cleaned[0].best_pipeline
print("Addressing all actions:", best_pipeline_clean.score(X_valid, y_valid, ["Log Loss␣
↪Binary"]), "\n")
```

```
Only dropping errors: OrderedDict([('Log Loss Binary', 0.26989110176765374)])

Addressing all actions: OrderedDict([('Log Loss Binary', 0.2661190195742235)])
```

We can compare the differences in model performance when we address all action items (warnings and errors) in comparison to when we only address errors. While it isn't guaranteed that addressing all actions will always have better performance, we do recommend doing so since we only raise these issues when we believe the features have problems that could negatively impact or not benefit the search.

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
```

```
EvalML version: 0.48.0
EvalML installation directory: /home/docs/checkouts/readthedocs.org/user_builds/feature-
↪labs-inc-evalml/envs/v0.48.0/lib/python3.8/site-packages/evalml

SYSTEM INFO
-----------
python: 3.8.6.final.0
python-bits: 64
OS: Linux
OS-release: 5.4.0-1045-aws
machine: x86_64
processor: x86_64
byteorder: little
LC_ALL: None
LANG: C.UTF-8
LOCALE: en_US.UTF-8

INSTALLED VERSIONS
------------------
tifffile: 2022.3.25
regex: 2022.3.15
distributed: 2022.3.0
dask: 2022.3.0
fsspec: 2022.2.0
sphinx-inline-tabs: 2022.1.2b11
pytz: 2022.1
certifi: 2021.10.8
setuptools: 61.2.0
types-setuptools: 57.4.11
pyzmq: 22.3.0
pip: 22.0.4
pyaml: 21.10.1
attrs: 21.4.0
argon2-cffi: 21.3.0
```

(continues on next page)

(continued from previous page)

```
packaging: 21.3
argon2-cffi-bindings: 21.2.0
pillow: 9.0.1
vowpalwabbit: 9.0.1
ipython: 8.2.0
click: 8.0.4
tenacity: 8.0.1
ipywidgets: 7.7.0
jupyter-client: 7.1.2
ipykernel: 6.10.0
notebook: 6.4.10
nbconvert: 6.4.5
tornado: 6.1
pyyaml: 6.0
psutil: 5.9.0
importlib-resources: 5.6.0
plotly: 5.6.0
nbformat: 5.2.0
traitlets: 5.1.1
tqdm: 4.63.1
fonttools: 4.31.2
importlib-metadata: 4.11.3
beautifulsoup4: 4.10.0
jupyter-core: 4.9.2
pexpect: 4.8.0
sphinx: 4.5.0
decorator: 4.4.2
jsonschema: 4.4.0
bleach: 4.1.0
zipp: 3.7.0
nltk: 3.7
widgetsnbextension: 3.6.0
matplotlib: 3.5.1
lightgbm: 3.3.2
idna: 3.3
jinja2: 3.1.1
threadpoolctl: 3.1.0
prompt-toolkit: 3.0.28
pyparsing: 3.0.7
requests: 2.27.1
pycparser: 2.21
imageio: 2.16.1
pygments: 2.11.2
babel: 2.9.1
python-dateutil: 2.8.2
astroid: 2.6.6
networkx: 2.5.1
category-encoders: 2.4.0
sortedcontainers: 2.4.0
convertdate: 2.4.0
soupsieve: 2.3.1
nlp-primitives: 2.3.0
```

(continues on next page)

(continued from previous page)

```
hijri-converter: 2.2.3
snowballstemmer: 2.2.0
readthedocs-sphinx-ext: 2.1.5
markupsafe: 2.1.1
zict: 2.1.0
charset-normalizer: 2.0.12
asttokens: 2.0.5
markdown-it-py: 2.0.1
sphinxcontrib-htmlhelp: 2.0.0
cloudpickle: 2.0.0
urllib3: 1.26.9
numpy: 1.21.5
six: 1.16.0
cffi: 1.15.0
wrapt: 1.12.1
pmdarima: 1.8.5
sphinx-autoapi: 1.8.3
send2trash: 1.8.0
scipy: 1.7.3
lazy-object-proxy: 1.7.1
featuretools: 1.7.0
tblib: 1.7.0
texttable: 1.6.4
debugpy: 1.6.0
nest-asyncio: 1.5.4
xgboost: 1.5.2
pandocfilters: 1.5.0
kiwisolver: 1.4.2
pandas: 1.4.1
unidecode: 1.3.4
imagesize: 1.3.0
pywavelets: 1.3.0
deprecated: 1.2.13
sphinxcontrib-websupport: 1.2.4
partd: 1.2.0
sphinxcontrib-serializinghtml: 1.1.5
joblib: 1.1.0
jupyterlab-widgets: 1.1.0
catboost: 1.0.4
msgpack: 1.0.3
sphinxcontrib-qthelp: 1.0.3
sphinxcontrib-applehelp: 1.0.2
sphinxcontrib-devhelp: 1.0.2
scikit-learn: 1.0.2
heapdict: 1.0.1
mock: 1.0.1
sphinxcontrib-jsmath: 1.0.1
numba: 0.55.1
evalml: 0.48.0
shap: 0.40.0
llvmlite: 0.38.0
wheel: 0.37.1
```

(continues on next page)

(continued from previous page)

```
cython: 0.29.28
scikit-image: 0.19.2
graphviz: 0.19.1
future: 0.18.2
jedi: 0.18.1
pyrsistent: 0.18.1
myst-parser: 0.16.1
docutils: 0.16
woodwork: 0.15.0
terminado: 0.13.3
statsmodels: 0.13.2
prometheus-client: 0.13.1
holidays: 0.13
toolz: 0.11.2
seaborn: 0.11.2
sktime: 0.11.0
cycler: 0.11.0
imbalanced-learn: 0.9.0
scikit-optimize: 0.9.0
nbsphinx: 0.8.8
mistune: 0.8.4
parso: 0.8.3
executing: 0.8.3
pydata-sphinx-theme: 0.8.1
commonmark: 0.8.1
alabaster: 0.7.12
pickleshare: 0.7.5
defusedxml: 0.7.1
ptyprocess: 0.7.0
testpath: 0.6.0
nbclient: 0.5.13
pymeeus: 0.5.11
patsy: 0.5.2
webencodings: 0.5.1
requirements-parser: 0.5.0
recommonmark: 0.5.0
colorama: 0.4.4
sphinx-rtd-theme: 0.4.3
entrypoints: 0.4
sphinx-copybutton: 0.4.0
mdit-py-plugins: 0.3.0
wcwidth: 0.2.5
pure-eval: 0.2.2
locket: 0.2.1
kaleido: 0.2.1
korean-lunar-calendar: 0.2.1
lime: 0.2.0.1
backcall: 0.2.0
ipython-genutils: 0.2.0
stack-data: 0.2.0
matplotlib-inline: 0.1.3
jupyterlab-pygments: 0.1.2
```

(continues on next page)

(continued from previous page)

```
mdurl: 0.1.0
slicer: 0.0.7
```

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 with time. 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.

[1]:
```
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%
11.0    1.40%
13.0    1.32%
12.5    1.21%
10.5    1.21%
        ...
0.2     0.03%
24.0    0.03%
```

(continues on next page)

(continued from previous page)

```
25.2    0.03%
22.7    0.03%
21.6    0.03%
Name: Temp, Length: 229, dtype: object
```

[2]:
```
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

[3]:
```
import plotly.graph_objects as go
```

[4]:
```
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 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`.

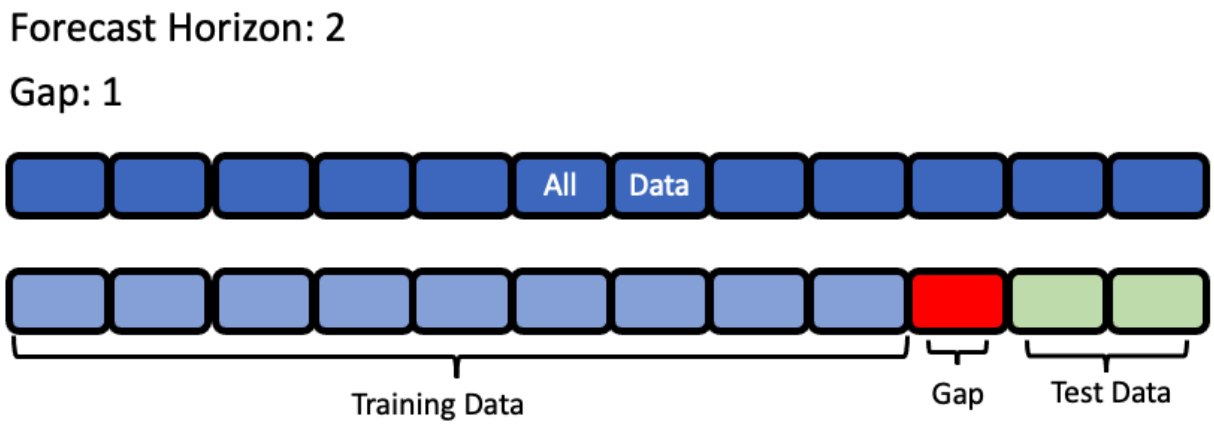
The `problem_configuration` is a dictionary specifying the following values:

- **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. Currently, this parameter is only used by some time-series specific models so in this example, we are passing in `None`.

Note that the values of these parameters must be in the same units as the training/testing data.

Visualization of forecast horizon and gap



[5]:
```
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,
                      allowed_model_families=["xgboost", "random_forest", "linear_model",
↪ "extra_trees",
                                              "decision_tree"]
                      )
```

```
/home/docs/checkouts/readthedocs.org/user_builds/feature-labs-inc-evalml/envs/v0.48.0/
↪lib/python3.8/site-packages/evalml/automl/automl_search.py:477: UserWarning:

Time series support in evalml is still in beta, which means we are still actively␣
↪building its core features. Please be mindful of that when running search().
```

[6]:
```
automl.search()
```

```
/home/docs/checkouts/readthedocs.org/user_builds/feature-labs-inc-evalml/envs/v0.48.0/
↪lib/python3.8/site-packages/sklearn/linear_model/_coordinate_descent.py:647:␣
↪ConvergenceWarning:

Objective did not converge. You might want to increase the number of iterations, check␣
↪the scale of the features or consider increasing regularisation. Duality gap: 3.
↪874e+03, tolerance: 1.703e+00
```

(continues on next page)

(continued from previous page)

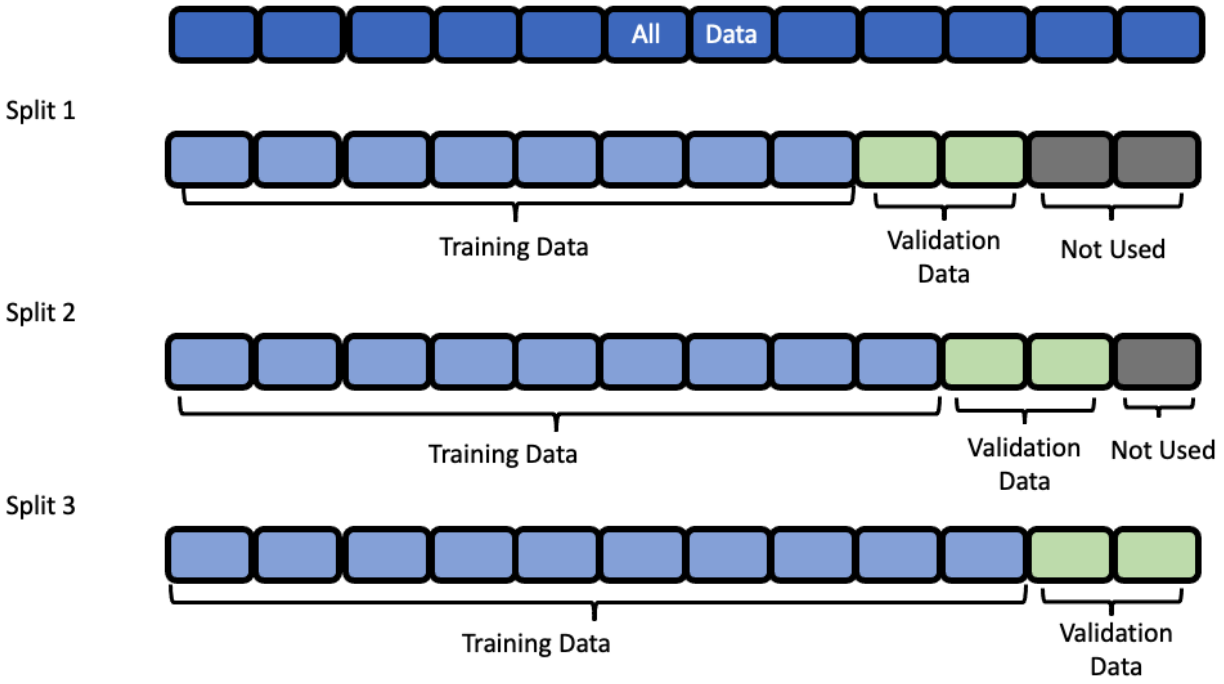
```
/home/docs/checkouts/readthedocs.org/user_builds/feature-labs-inc-evalml/envs/v0.48.0/
↪lib/python3.8/site-packages/sklearn/linear_model/_coordinate_descent.py:647:␣
↪ConvergenceWarning:

Objective did not converge. You might want to increase the number of iterations, check␣
↪the scale of the features or consider increasing regularisation. Duality gap: 7.
↪306e+03, tolerance: 2.964e+00

/home/docs/checkouts/readthedocs.org/user_builds/feature-labs-inc-evalml/envs/v0.48.0/
↪lib/python3.8/site-packages/sklearn/linear_model/_coordinate_descent.py:647:␣
↪ConvergenceWarning:

Objective did not converge. You might want to increase the number of iterations, check␣
↪the scale of the features or consider increasing regularisation. Duality gap: 1.
↪040e+04, tolerance: 4.094e+00
```

4.10.5 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. 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.

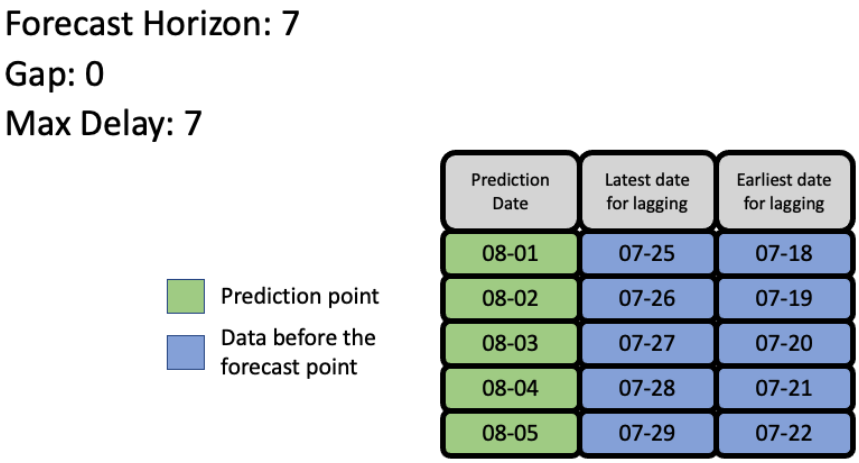
[7]:
```
import pandas as pd
baseline = automl.get_pipeline(0)
baseline.fit(X_train, y_train)
naive_baseline_preds = baseline.predict_in_sample(X_test, y_test, objective=None,
                                                  X_train=X_train, y_train=y_train)
expected_preds = pd.concat([y_train.iloc[-7:], y_test]).shift(7).iloc[7:]
pd.testing.assert_series_equal(expected_preds, naive_baseline_preds)
```

Feature Engineering

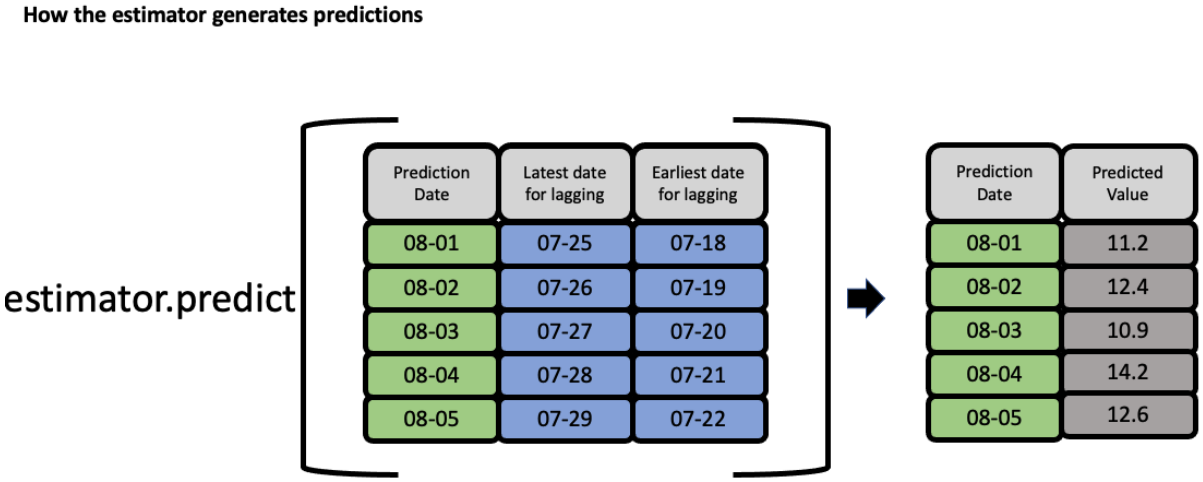
EvalML uses the values of `gap`, `forecast_horizon`, and `max_delay` to calculate a "window" of allowed dates that can be used for engineering the features of each row in the validation/test set. The formula for computing the bounds of the window is:

[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*

4.10.6 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.

[8]:
```
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
↪']
```

[9]:
```
best_pipeline_score
```

[9]:
```
1.5252435247344698
```

[10]:
```
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']
```

[11]:
```
naive_baseline_score
```

[11]:
```
2.3
```

The pipeline found by AutoMLSearch has a 31% improvement over the naive forecast!

[12]:
```
automl.objective.calculate_percent_difference(best_pipeline_score, naive_baseline_score)
```

[12]:
```
33.68506414197957
```

4.10.7 Visualize the predictions over time

[13]:
```
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

4.10.8 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

```
[14]: pl.predict_in_sample(X_test, y_test, objective=None, X_train=X_train, y_train=y_train)
```

```
[14]: 3287    14.461731
      3288    14.405880
      3289    14.639265
      3290    14.118989
      3291    14.074054
                ...
      3647    14.031675
      3648    13.953069
      3649    13.992977
      3650    14.033945
      3651    13.706983
      Name: Temp, Length: 365, dtype: float64
```

4.10.9 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 sunseen 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`.

```
[15]: 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
      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

```
[16]: pl.predict(X_test.iloc[:pl.forecast_horizon], objective=None, X_train=X_train, y_train=y_
      ↪train)
```

```
[16]: 3287    14.461731
      3288    14.405880
      3289    14.639265
      3290    14.118989
      3291    14.074054
      3292    14.129104
      3293    14.340058
      Name: Temp, dtype: float64
```

predict – Test set size is less than forecast horizon

[17]:
```
pl.predict(X_test.iloc[:pl.forecast_horizon - 2], objective=None, X_train=X_train, y_
↪train=y_train)
```

[17]:
```
3287    14.461731
3288    14.405880
3289    14.639265
3290    14.118989
3291    14.074054
Name: Temp, dtype: float64
```

predict – Test set size index starts at 0

[18]:
```
pl.predict(X_test.iloc[:pl.forecast_horizon].reset_index(drop=True), objective=None, X_
↪train=X_train, y_train=y_train)
```

[18]:
```
3287    14.461731
3288    14.405880
3289    14.639265
3290    14.118989
3291    14.074054
3292    14.129104
3293    14.340058
Name: Temp, dtype: float64
```

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.

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.
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.
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.
DFSTransformer	Featuretools DFS component that generates features for the input features.
PolynomialDetrender	Removes trends from time series by fitting a polynomial 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.

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 or Pearson correlation.
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*, *custom_hyperparameters=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.
- **custom_hyperparameters** (*dict*) – Custom hyperparameter 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.
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]

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*, *pipeline_params=None*, *custom_hyperparameters=None*, *n_jobs=- 1*, *text_in_ensembling=False*, *top_n=3*, *num_long_explore_pipelines=50*, *num_long_pipelines_per_batch=10*, *allow_long_running_models=False*, *features=None*, *verbose=False*)

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.
- **pipeline_params** (*dict or None*) – Pipeline-level parameters that should be passed to the proposed pipelines. Defaults to None.
- **custom_hyperparameters** (*dict or None*) – Custom hyperparameter ranges specified for pipelines to iterate over. 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.

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.
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)

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*, *pipeline_params=None*, *custom_hyperparameters=None*, *_estimator_family_order=None*, *allow_long_running_models=False*, *features=None*, *verbose=False*)

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.
- **pipeline_params** (*dict or None*) – Pipeline-level parameters that should be passed to the proposed pipelines. Defaults to None.
- **custom_hyperparameters** (*dict or None*) – Custom hyperparameter ranges specified for pipelines to iterate over. 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.

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.
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.

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*, *custom_hyperparameters=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.
- **custom_hyperparameters** (*dict*) – Custom hyperparameter 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.
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]

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*, *pipeline_params=None*, *custom_hyperparameters=None*, *n_jobs=- 1*, *text_in_ensembling=False*, *top_n=3*, *num_long_explore_pipelines=50*, *num_long_pipelines_per_batch=10*, *allow_long_running_models=False*, *features=None*, *verbose=False*)

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.
- **pipeline_params** (*dict or None*) – Pipeline-level parameters that should be passed to the proposed pipelines. Defaults to None.
- **custom_hyperparameters** (*dict or None*) – Custom hyperparameter ranges specified for pipelines to iterate over. 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.

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.
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)

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*, *pipeline_params=None*, *custom_hyperparameters=None*, *_estimator_family_order=None*, *allow_long_running_models=False*, *features=None*, *verbose=False*)

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.
- **pipeline_params** (*dict or None*) – Pipeline-level parameters that should be passed to the proposed pipelines. Defaults to None.
- **custom_hyperparameters** (*dict or None*) – Custom hyperparameter ranges specified for pipelines to iterate over. 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.

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.
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.

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*)

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.

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*)

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.

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*)

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*)

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*)

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.

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*)

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.

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*)

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.

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*)

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.

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*)

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.

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*)

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*)

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*)

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.

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*)

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.

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*)

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.

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*, *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*, *pipeline_parameters=None*, *custom_hyperparameters=None*, *sampler_method='auto'*, *sampler_balanced_ratio=0.25*, *allow_long_running_models=False*, *_ensembling_split_size=0.2*, *_pipelines_per_batch=5*, *automl_algorithm='default'*, *engine='sequential'*, *verbose=False*)

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.
- **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.
- **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.
- **pipeline_parameters** (*dict*) – A dict of the parameters used to initialize a pipeline with. Keys should consist of the component names and values should specify parameter values

  e.g. pipeline_parameters = { 'Imputer' : { 'numeric_impute_strategy': 'most_frequent' } }
- **custom_hyperparameters** (*dict*) – A dict of the hyperparameter ranges used to iterate over during search. Keys should consist of the component names and values should specify a singular value or skopt.Space.

  e.g. custom_hyperparameters = { 'Imputer' : { 'numeric_impute_strategy': Categorical(['most_frequent', 'median']) } }
- **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.

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*, *show_iteration_plot=True*)

Find the best pipeline for the data set.

Parameters **show_iteration_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.

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*)

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.

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*, ***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.
- ****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.

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.
make_data_splitter	Given the training data and ML problem parameters, compute a data splitting method to use during AutoML search.
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.**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.**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.
SequentialEngine	The default engine for the AutoML search.

Functions

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.
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*, *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*, *pipeline_parameters=None*, *custom_hyperparameters=None*, *sampler_method='auto'*, *sampler_balanced_ratio=0.25*, *allow_long_running_models=False*, *_ensembling_split_size=0.2*, *_pipelines_per_batch=5*, *automl_algorithm='default'*, *engine='sequential'*, *verbose=False*)

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.
- **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.
- **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.
- **pipeline_parameters** (*dict*) – A dict of the parameters used to initialize a pipeline with. Keys should consist of the component names and values should specify parameter values

  e.g. pipeline_parameters = { 'Imputer' : { 'numeric_impute_strategy': 'most_frequent' } }

- **custom_hyperparameters** (*dict*) – A dict of the hyperparameter ranges used to iterate over during search. Keys should consist of the component names and values should specify a singular value or skopt.Space.

  e.g. custom_hyperparameters = { 'Imputer' : { 'numeric_impute_strategy': Categorical(['most_frequent', 'median']) } }

- **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.

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*, *show_iteration_plot=True*)

Find the best pipeline for the data set.

Parameters **show_iteration_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.

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*)

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*)

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.**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.**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*)

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.

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*, ***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.
- ****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*)

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.

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]",
```

(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 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.
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 data check.
- **message_code** (*DataCheckMessageCode*) – Message code associated with message. Defaults to None.
- **details** (*dict*) – Additional useful information 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 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_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.
HAS_ID_COLUMN	Message code for data that has ID columns.
HAS_OUTLIERS	Message code for when outliers are detected.

continues on next page

Table 1 – continued from previous page

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.
TIMESERIES_PARAMETERS_NOT_COMPATIBLE_WITH_SPLIT	Message code when the time series parameters are too large for the smallest data split.
TIMESERIES_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'*)

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".

Methods

name	Return a name describing the data check.
validate	Checks if the target data has equal intervals and is sorted.

name(*cls*)

Return a name describing the data check.

validate(*self*, *X*, *y*)

Checks if the target data has equal intervals and is sorted.

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 dates with hourly frequency appended to the end of a series of days, which is inconsistent with the frequency of the previous 9 dates (1 day).

```
>>> X = pd.DataFrame(pd.date_range("2021-01-01", periods=6).append(pd.date_
↪range("2021-01-07", periods=3, freq="H")), columns=["dates"])
>>> y = pd.Series([0, 1, 0, 1, 1, 0, 0, 0, 1, 0])
>>> 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": "No frequency could be detected in column 'dates',␣
↪possibly due to uneven intervals.",
```

(continues on next page)

(continued from previous page)

```
...         "data_check_name": "DateTimeFormatDataCheck",
...         "level": "error",
...         "code": "DATETIME_HAS_UNEVEN_INTERVALS",
...         "details": {"columns": None, "rows": None},
...         "action_options": []
...     }
... ]
```

The column "dates" has the date 2021-01-31 appended to the end, 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=1)), columns=["dates"])
>>> y = pd.Series([0, 1, 0, 1, 1, 0, 0, 0, 1, 0])
>>> 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": []
...     }
... ]
```

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])
>>> 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": []
...     }
... ]
```

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) == [
...     {
```

(continues on next page)

(continued from previous page)

```
...         "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) == [
...     {
...         "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"]]
>>> dates[0][0] = None
>>> df = pd.DataFrame(dates, columns=["days", "days2"])
>>> 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",
```

(continues on next page)

(continued from previous page)

```
...         "details": {"columns": None, "rows": None},
...         "code": "DATETIME_HAS_NAN",
...         "action_options": []
...     }
... ]
...
```

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.
- **datetime_column** (*str*) – The name of the column containing datetime information to be used for time series problems.
- **y.** (*Default to "index" indicating that the datetime information is in the index of X or*) –

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({
...     "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({
...     "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}
...             }
...         ]
...     }
```

(continues on next page)

(continued from previous page)

```
... ]
```

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*)

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.

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")
>>> assert target_check.validate(X, y) == [
...     {
...         "message": "Target is unsupported Unknown type. Valid Woodwork␣
↪logical types include: integer, double, boolean, integer_nullable, boolean_
↪nullable, age_nullable",
...         "data_check_name": "InvalidTargetDataCheck",
...         "level": "error",
...         "details": {"columns": None, "rows": None, "unsupported_type":
↪"unknown"},
...         "code": "TARGET_UNSUPPORTED_TYPE",
```

(continues on next page)

(continued from previous page)

```
...         "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",
...         "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": None,
...             "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": [],
...             "indices_not_in_target": [0]
```

(continues on next page)

(continued from previous page)

```
...         },
...         "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])
>>> X = pd.DataFrame({"col_1": col, "col_2": col * 3})
>>> y = pd.Series([1, 0, 0, 1, 0])
...
>>> 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])
...
>>> novar_dc = NoVarianceDataCheck()
>>> assert novar_dc.validate(X, y) == [
...     {
...         "message": "'First_Column' has 1 unique value.",
...         "data_check_name": "NoVarianceDataCheck",
...         "level": "error",
...         "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": "error",
...         "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": "error",
...         "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": "error",
...         "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": "error",
...         "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) == [
```

(continues on next page)

(continued from previous page)

```
...     {
...         "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": []
...     }
... ]
```

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_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_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}
```

(continues on next page)

(continued from previous page)

```
...             }
...         ]
...     },
...     {
...         "message": "Column(s) 'few_null' have 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"}
...                             }
...                         }
...                     }
...                 }
...             }
...         ]
...     }
... ]
```

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%.

```
>>> 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",
```

(continues on next page)

(continued from previous page)

```
...         "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},
...                 "parameters": {}
...             }
...         ]
...     },
...     {
...         "message": "Column(s) 'lots_of_null', 'few_null' have 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"}}
...                        }
...                    }
...                }
...             }
```

(continues on next page)

(continued from previous page)

```
...         ]
...     }
... ]
```

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]
```

(continues on next page)

(continued from previous page)

```
... })
>>> 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]}},
```

(continues on next page)

(continued from previous page)

```
...         "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) == [
...     {
```

(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

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
```

(continues on next page)

(continued from previous page)

```
...                 }
...             }
...         ]
...     }
... ]
...
>>> 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",
...         "data_check_name": "TargetDistributionDataCheck",
...         "level": "error",
...         "details": {"columns": None, "rows": None, "unsupported_type":
↪"datetime"},
...         "code": "TARGET_UNSUPPORTED_TYPE",
...         "action_options": []
...     }
... ]
```

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 or Pearson correlation.

Contents

class evalml.data_checks.target_leakage_data_check.**TargetLeakageDataCheck**(*pct_corr_threshold=0.95*, *method='mutual'*)

Check if any of the features are highly correlated with the target by using mutual information or Pearson correlation.

If *method='mutual'*, this data check uses mutual information and supports all target and feature types. Otherwise, if *method='pearson'*, it uses Pearson correlation and only supports binary with numeric and boolean dtypes. Pearson correlation returns a value in [-1, 1], while mutual information returns a value in [0, 1].

Parameters

- **pct_corr_threshold** (*float*) – The correlation threshold to be considered leakage. Defaults to 0.95.
- **method** (*string*) – The method to determine correlation. Use 'mutual' for mutual information, otherwise 'pearson' for Pearson correlation. Defaults to 'mutual'.

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 or Pearson 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 or Pearson correlation.

If *method='mutual'*, supports all target and feature types. Otherwise, if *method='pearson'* only supports binary with numeric and boolean dtypes. Pearson correlation returns a value in [-1, 1], while mutual information returns a value in [0, 1].

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],
...     "x": [42, 54, 12, 64, 12],
...     "y": [13, 5, 13, 74, 24],
... })
>>> y = pd.Series([10, 42, 31, 51, 40])
...
>>> 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",
```

(continues on next page)

(continued from previous page)

```
...         "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 information.

```
>>> 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"}
>>> target_leakage_check = TimeSeriesParametersDataCheck(problem_
↪configuration=problem_config, n_splits=4)
>>> assert target_leakage_check.validate(X, y) == [
```

(continues on next page)

(continued from previous page)

```
...     {
...         "message": "Since the data has 100 observations and n_splits=4, the␣
↪smallest "
...                    "split would have 20 observations. Since 21 (gap + max_
↪delay + forecast_horizon)"
...                    " >= 20, 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": {
...             "columns": None,
...             "rows": None,
...             "max_window_size": 21,
...             "min_split_size": 20,
...             "n_obs": 100,
...             "n_splits": 4
...         },
...         "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": []
```

(continues on next page)

(continued from previous page)

```
...     }
... ]
```

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}
```

(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

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 or Pearson correlation.
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.
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 data check.
- **message_code** (DataCheckMessageCode) – Message code associated with message. Defaults to None.
- **details** (*dict*) – Additional useful information 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 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_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.
HAS_ID_COLUMN	Message code for data that has ID columns.
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.

continues on next page

Table 2 – continued from previous page

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.
TIMESERIES_PARAMETERS_NOT_COMPATIBLE_WITH_SPLIT	Message code when the time series parameters are too large for the smallest data split.
TIMESERIES_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'*)

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".

Methods

name	Return a name describing the data check.
validate	Checks if the target data has equal intervals and is sorted.

name(*cls*)

Return a name describing the data check.

validate(*self*, *X*, *y*)

Checks if the target data has equal intervals and is sorted.

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 dates with hourly frequency appended to the end of a series of days, which is inconsistent with the frequency of the previous 9 dates (1 day).

```
>>> X = pd.DataFrame(pd.date_range("2021-01-01", periods=6).append(pd.date_
↪range("2021-01-07", periods=3, freq="H")), columns=["dates"])
>>> y = pd.Series([0, 1, 0, 1, 1, 0, 0, 0, 1, 0])
>>> 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": "No frequency could be detected in column 'dates',␣
↪possibly due to uneven intervals.",
...         "data_check_name": "DateTimeFormatDataCheck",
...         "level": "error",
...         "code": "DATETIME_HAS_UNEVEN_INTERVALS",
...         "details": {"columns": None, "rows": None},
...         "action_options": []
...      }
... ]
```

The column "dates" has the date 2021-01-31 appended to the end, 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=1)), columns=["dates"])
>>> y = pd.Series([0, 1, 0, 1, 1, 0, 0, 0, 1, 0])
>>> 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": []
```

(continues on next page)

(continued from previous page)

```
...         }
... ]
```

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])
>>> 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": []
...     }
... ]
```

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) == [
...     {
...         "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"]]
>>> dates[0][0] = None
>>> df = pd.DataFrame(dates, columns=["days", "days2"])
>>> 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": []
...     }
... ]
...
```

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.
- **datetime_column** (*str*) – The name of the column containing datetime information to be used for time series problems.
- **y.** (*Default to "index" indicating that the datetime information is in the index of X or*) –

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({
...     "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({
...     "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}
...             }
...         ]
...     }
```

(continues on next page)

(continued from previous page)

```
... ]
```

class evalml.data_checks.**InvalidTargetDataCheck**(*problem_type*, *objective*, *n_unique=100*)

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.

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")
>>> assert target_check.validate(X, y) == [
...     {
...         "message": "Target is unsupported Unknown type. Valid Woodwork␣
↪logical types include: integer, double, boolean, 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",
...         "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",
```

(continues on next page)

(continued from previous page)

```
...         "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": None,
...             "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},
```

(continues on next page)

(continued from previous page)

```
...         "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": []
...     }
... ]
```

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])
>>> X = pd.DataFrame({"col_1": col, "col_2": col * 3})
>>> y = pd.Series([1, 0, 0, 1, 0])
...
>>> 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": []
...     }
... ]
```

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": "error",
...         "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": "error",
...         "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",
```

(continues on next page)

(continued from previous page)

```
...         "level": "error",
...         "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": "error",
...         "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": "error",
...         "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": [
```

(continues on next page)

(continued from previous page)

```
...             {
...                 "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_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_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",
```

(continues on next page)

(continued from previous page)

```
...         "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 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"}
...                             }
...                         }
...                     }
...                 }
...             }
...         ]
...     }
... ]
```

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%.

```
>>> highly_null_dc = NullDataCheck(pct_null_row_threshold=0.50)
```

(continues on next page)

(continued from previous page)

```
>>> 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},
...                 "parameters": {}
...             }
...         ]
...     },
...     {
...         "message": "Column(s) 'lots_of_null', 'few_null' have 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",
```

(continues on next page)

(continued from previous page)

```
...                 "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)
```

(continues on next page)

(continued from previous page)

```
>>> 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": [
...             {
```

(continues on next page)

(continued from previous page)

```
...                 "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) == [
...     {
...         "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) == []
...
```

(continues on next page)

(continued from previous page)

```
...
>>> 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",
...         "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='mutual'*)

Check if any of the features are highly correlated with the target by using mutual information or Pearson correlation.

If *method='mutual'*, this data check uses mutual information and supports all target and feature types. Otherwise, if *method='pearson'*, it uses Pearson correlation and only supports binary with numeric and boolean dtypes. Pearson correlation returns a value in [-1, 1], while mutual information returns a value in [0, 1].

Parameters

- **pct_corr_threshold** (*float*) – The correlation threshold to be considered leakage. Defaults to 0.95.
- **method** (*string*) – The method to determine correlation. Use 'mutual' for mutual information, otherwise 'pearson' for Pearson correlation. Defaults to 'mutual'.

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 or Pearson 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 or Pearson correlation.

If *method='mutual'*, supports all target and feature types. Otherwise, if *method='pearson'* only supports binary with numeric and boolean dtypes. Pearson correlation returns a value in [-1, 1], while mutual information returns a value in [0, 1].

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],
...     "x": [42, 54, 12, 64, 12],
...     "y": [13, 5, 13, 74, 24],
... })
>>> y = pd.Series([10, 42, 31, 51, 40])
...
>>> 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 information.

```
>>> 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}
```

(continues on next page)

(continued from previous page)

```
...             }
...         ]
...     }
... ]
```

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"}
```

(continues on next page)

(continued from previous page)

```
>>> target_leakage_check = TimeSeriesParametersDataCheck(problem_
↪configuration=problem_config, n_splits=4)
>>> assert target_leakage_check.validate(X, y) == [
...     {
...         "message": "Since the data has 100 observations and n_splits=4, the␣
↪smallest "
...                    "split would have 20 observations. Since 21 (gap + max_
↪delay + forecast_horizon)"
...                    " >= 20, 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": {
...             "columns": None,
...             "rows": None,
...             "max_window_size": 21,
...             "min_split_size": 20,
...             "n_obs": 100,
...             "n_splits": 4
...         },
...         "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)
```

(continues on next page)

(continued from previous page)

```
>>> 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": [
...             {
...                 "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.
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.

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.**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.
ValidationErrorCode	Enum identifying the type of error encountered in holdout 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.

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_PARTIAL_DEPENDENCE_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.**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* *include_explain* *metric=None*, *output_format='* *callback=None*, *training_data=No* *training_target=N* *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.

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]]

graphs

Model understanding graphing utilities.

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.
confusion_matrix	Confusion matrix for binary and multiclass classification.
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_confusion_matrix	Generate and display a confusion matrix 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.
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.
visualize_decision_tree	Generate an image visualizing the decision tree.

Contents

evalml.model_understanding.graphs.**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.graphs.**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.graphs.**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.graphs.**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.graphs.**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.graphs.**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.graphs.**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.graphs.**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.graphs.**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.graphs.**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.graphs.**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.graphs.**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.graphs.**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.graphs.**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.graphs.**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.graphs.**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.graphs.**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.graphs.**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 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.graphs.**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.

Args: 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.graphs.**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.

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'*)

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.

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.

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_featu*
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.

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_influential_features	Finds the most influential features as well as any detrimental features from a dataframe of feature importances.
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.
readable_explanation	Outputs a human-readable explanation of trained pipeline behavior.
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_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.**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'*)

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.

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.**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.

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 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.

Args: 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.

Args: 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.

Args: 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.

Args: 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.

Args: 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.
MedianAE	Median absolute error for regression.
MSE	Mean squared error for regression.
Precision	Precision score for binary classification.

continues on next page

Table 3 – continued from previous page

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_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.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_like_percentage	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_like_percentage	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_percentage	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_percentage	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_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.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_like_percentage	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.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_like_percentage	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_percentage	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), 18.8393325)
```

Attributes

expected_range	[0, 1]
greater_is_better	False
is_bounded_like_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_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_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.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_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.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_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.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_percentage	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_like_percentage	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_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.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_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.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_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_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.

Args: 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.

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.

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.
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.

continues on next page

Table 4 – continued from previous page

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.
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.

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_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.**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_like_percentage	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_like_percentage	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_percentage	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_percentage	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.

Args: 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_like_percentage	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_percentage	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_percentage	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.

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_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_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), 18.8393325)
```

Attributes

expected_range	[0, 1]
greater_is_better	False
is_bounded_like_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.**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_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_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_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.**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_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.

Args: 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.

Args: 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_percentage	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_percentage	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.

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_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.**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.

Args: 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.
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.

clone(*self*)

Constructs a new component with the same parameters 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*, *y=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

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*)

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*)

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.

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,
ran-
dom_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 == {
...     '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

hyperparameter_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.
load	Loads component at file path.
needs_fitting	Returns boolean determining 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.

clone(*self*)

Constructs a new component with the same parameters 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*, *y=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

static load(*file_path*)

Loads component at file path.

Parameters **file_path** (*str*) – Location to load file.

Returns ComponentBase object

needs_fitting(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need 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 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*)

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.

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,
ran-
dom_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",
...     ],
... }
...
>>> cg = ComponentGraph(component_graph)
>>> assert cg.default_parameters == {
```

(continues on next page)

(continued from previous page)

```
...     'Random Forest Regressor': {'n_estimators': 100,
...                                 'max_depth': 6,
...                                 'n_jobs': -1},
...     'Stacked Ensemble Regressor': {'final_estimator': ElasticNetRegressor,
...                                    'n_jobs': -1}}
```

Attributes

hyperparameter_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.
load	Loads component at file path.
needs_fitting	Returns boolean determining 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.

clone(*self*)

Constructs a new component with the same parameters 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*, *y=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

static load(*file_path*)

Loads component at file path.

Parameters **file_path** (*str*) – Location to load file.

Returns ComponentBase object

needs_fitting(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need 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 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*)

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.

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.
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.

clone(*self*)

Constructs a new component with the same parameters 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*, *y=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

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*)

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*)

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.

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 = {
...     "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

hyperparameter_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.
load	Loads component at file path.
needs_fitting	Returns boolean determining 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.

clone(*self*)

Constructs a new component with the same parameters 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*, *y=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

static load(*file_path*)

Loads component at file path.

Parameters **file_path** (*str*) – Location to load file.

Returns ComponentBase object

needs_fitting(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need 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 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*)

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.

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": [
...         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

hyperparameter_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.
load	Loads component at file path.
needs_fitting	Returns boolean determining 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.

clone(*self*)

Constructs a new component with the same parameters 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*, *y=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

static load(*file_path*)

Loads component at file path.

Parameters **file_path** (*str*) – Location to load file.

Returns ComponentBase object

needs_fitting(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need 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 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*)

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.

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=*
ran-
dom_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.
load	Loads component at file path.
needs_fitting	Returns boolean determining 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.

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.

static load(*file_path*)

Loads component at file path.

Parameters **file_path** (*str*) – Location to load file.

Returns ComponentBase object

needs_fitting(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need 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.

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

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.
load	Loads component at file path.
needs_fitting	Returns boolean determining 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 probability estimates for labels.
save	Saves component at file path.

clone(*self*)

Constructs a new component with the same parameters and random state.

Returns A new instance of this component with identical parameters and random state.

default_parameters(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

Returns Default parameters for this component.

Return type dict

describe(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

Parameters

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description 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

static load(*file_path*)

Loads component at file path.

Parameters **file_path** (*str*) – Location to load file.

Returns ComponentBase object

needs_fitting(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need 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.DataFrame

predict_proba(*self*, *X*)

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.

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

hyperparameter_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.
load	Loads component at file path.
needs_fitting	Returns boolean determining 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.

clone(*self*)

Constructs a new component with the same parameters and random state.

Returns A new instance of this component with identical parameters and random state.

default_parameters(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

Returns Default parameters for this component.

Return type dict

describe(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

Parameters

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description 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.

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 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

static load(*file_path*)

Loads component at file path.

Parameters **file_path** (*str*) – Location to load file.

Returns ComponentBase object

needs_fitting(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need 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 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*)

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.

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**(*penalty='elasticnet'*, *C=1.0*, *l1_ratio=0.15*, *multi_class='auto'*, *solver='saga'*, *n_jobs=-1*, *random_state=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

hyperparameter_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.
load	Loads component at file path.
needs_fitting	Returns boolean determining 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.

clone(*self*)

Constructs a new component with the same parameters and random state.

Returns A new instance of this component with identical parameters and random state.

default_parameters(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

Returns Default parameters for this component.

Return type dict

describe(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

Parameters

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description 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

static load(*file_path*)

Loads component at file path.

Parameters **file_path** (*str*) – Location to load file.

Returns ComponentBase object

needs_fitting(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need 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 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*)

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.

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_fr
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

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.
load	Loads component at file path.
needs_fitting	Returns boolean determining 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.

clone(*self*)

Constructs a new component with the same parameters and random state.

Returns A new instance of this component with identical parameters and random state.

default_parameters(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

Returns Default parameters for this component.

Return type dict

describe(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

Parameters

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description 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.

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 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

static load(*file_path*)

Loads component at file path.

Parameters **file_path** (*str*) – Location to load file.

Returns ComponentBase object

needs_fitting(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need 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 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*)

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.

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-
rithm
*leaf_*
p=2
ran-
dom
***kw*

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

hyperparameter_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.
load	Loads component at file path.
needs_fitting	Returns boolean determining 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.

clone(*self*)

Constructs a new component with the same parameters and random state.

Returns A new instance of this component with identical parameters and random state.

default_parameters(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

Returns Default parameters for this component.

Return type dict

describe(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

Parameters

- **print_name** (*bool, optional*) – whether to print name of component

- **return_dict** (*bool, optional*) – whether to return description 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*, *y=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

static load(*file_path*)

Loads component at file path.

Parameters **file_path** (*str*) – Location to load file.

Returns ComponentBase object

needs_fitting(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need 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 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*)

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.

lightgbm_classifier

LightGBM Classifier.

Module Contents

Classes Summary

LightGBMClassifier	LightGBM Classifier.

Contents

class evalml.pipelines.components.estimators.classifiers.lightgbm_classifier.**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

hyperparameter_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.
load	Loads component at file path.
needs_fitting	Returns boolean determining 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.

clone(*self*)

Constructs a new component with the same parameters and random state.

Returns A new instance of this component with identical parameters and random state.

default_parameters(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

Returns Default parameters for this component.

Return type dict

describe(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

Parameters

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description 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.

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

static load(*file_path*)

Loads component at file path.

Parameters **file_path** (*str*) – Location to load file.

Returns ComponentBase object

needs_fitting(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need 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.

logistic_regression_classifier

Logistic Regression Classifier.

Module Contents

Classes Summary

LogisticRegressionClassifier	Logistic Regression Classifier.

Contents

class evalml.pipelines.components.estimators.classifiers.logistic_regression_classifier.**LogisticRegress**

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.
load	Loads component at file path.
needs_fitting	Returns boolean determining 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.

clone(*self*)

Constructs a new component with the same parameters and random state.

Returns A new instance of this component with identical parameters and random state.

default_parameters(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

Returns Default parameters for this component.

Return type dict

describe(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

Parameters

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description 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*, *y=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

static load(*file_path*)

Loads component at file path.

Parameters **file_path** (*str*) – Location to load file.

Returns ComponentBase object

needs_fitting(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need 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 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*)

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.

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-parame-ter_ranges	{ “n_estimators”: Integer(10, 1000), “max_depth”: Integer(1, 10),}
model_family	ModelFamily.RANDOM_FOREST
modi-fies_features	True
modi-fies_target	False
name	Random Forest Classifier
sup-ported_problem_types	[ProblemTypes.BINARY, ProblemTypes.MULTICLASS, Problem-Types.TIME_SERIES_BINARY, ProblemTypes.TIME_SERIES_MULTICLASS,]
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.
feature_importance	Returns importance associated with each feature.
fit	Fits estimator to 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.
predict	Make predictions using selected features.
predict_proba	Make probability estimates for labels.
save	Saves component at file path.

clone(*self*)

Constructs a new component with the same parameters and random state.

Returns A new instance of this component with identical parameters and random state.

default_parameters(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

Returns Default parameters for this component.

Return type dict

describe(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

Parameters

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description 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.

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 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

static load(*file_path*)

Loads component at file path.

Parameters **file_path** (*str*) – Location to load file.

Returns ComponentBase object

needs_fitting(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need 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 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*)

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.

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.
load	Loads component at file path.
needs_fitting	Returns boolean determining 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.

clone(*self*)

Constructs a new component with the same parameters and random state.

Returns A new instance of this component with identical parameters and random state.

default_parameters(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

Returns Default parameters for this component.

Return type dict

describe(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

Parameters

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description 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*, *y=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

static load(*file_path*)

Loads component at file path.

Parameters **file_path** (*str*) – Location to load file.

Returns ComponentBase object

needs_fitting(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need 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 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*)

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.

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

hyperparameter_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.
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.

clone(*self*)

Constructs a new component with the same parameters and random state.

Returns A new instance of this component with identical parameters and random state.

default_parameters(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

Returns Default parameters for this component.

Return type dict

describe(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

Parameters

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description 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*, *y=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

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*)

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*)

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.

class evalml.pipelines.components.estimators.classifiers.vowpal_wabbit_classifiers.**VowpalWabbitBinaryCl**

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.
load	Loads component at file path.
needs_fitting	Returns boolean determining 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.

clone(*self*)

Constructs a new component with the same parameters and random state.

Returns A new instance of this component with identical parameters and random state.

default_parameters(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

Returns Default parameters for this component.

Return type dict

describe(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

Parameters

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description 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*, *y=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

static load(*file_path*)

Loads component at file path.

Parameters **file_path** (*str*) – Location to load file.

Returns ComponentBase object

needs_fitting(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need 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 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*)

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.

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

hyperparameter_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.
load	Loads component at file path.
needs_fitting	Returns boolean determining 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.

clone(*self*)

Constructs a new component with the same parameters and random state.

Returns A new instance of this component with identical parameters and random state.

default_parameters(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

Returns Default parameters for this component.

Return type dict

describe(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

Parameters

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description 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*, *y=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

static load(*file_path*)

Loads component at file path.

Parameters **file_path** (*str*) – Location to load file.

Returns ComponentBase object

needs_fitting(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need 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 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*)

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.

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_estimators
ran-
dom_seed=0
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

hyperparameter_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.
load	Loads component at file path.
needs_fitting	Returns boolean determining 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.

clone(*self*)

Constructs a new component with the same parameters and random state.

Returns A new instance of this component with identical parameters and random state.

default_parameters(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

Returns Default parameters for this component.

Return type dict

describe(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

Parameters

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description 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

static load(*file_path*)

Loads component at file path.

Parameters **file_path** (*str*) – Location to load file.

Returns ComponentBase object

needs_fitting(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need 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.

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

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.
load	Loads component at file path.
needs_fitting	Returns boolean determining 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.

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.

static load(*file_path*)

Loads component at file path.

Parameters **file_path** (*str*) – Location to load file.

Returns ComponentBase object

needs_fitting(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need 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.

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

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.
load	Loads component at file path.
needs_fitting	Returns boolean determining 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 probability estimates for labels.
save	Saves component at file path.

clone(*self*)

Constructs a new component with the same parameters and random state.

Returns A new instance of this component with identical parameters and random state.

default_parameters(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

Returns Default parameters for this component.

Return type dict

describe(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

Parameters

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description 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

static load(*file_path*)

Loads component at file path.

Parameters **file_path** (*str*) – Location to load file.

Returns ComponentBase object

needs_fitting(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need 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.DataFrame

predict_proba(*self*, *X*)

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.

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

hyperparameter_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.
load	Loads component at file path.
needs_fitting	Returns boolean determining 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.

clone(*self*)

Constructs a new component with the same parameters and random state.

Returns A new instance of this component with identical parameters and random state.

default_parameters(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

Returns Default parameters for this component.

Return type dict

describe(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

Parameters

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description 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.

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 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

static load(*file_path*)

Loads component at file path.

Parameters **file_path** (*str*) – Location to load file.

Returns ComponentBase object

needs_fitting(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need 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 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*)

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.

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

hyperparameter_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.
load	Loads component at file path.
needs_fitting	Returns boolean determining 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.

clone(*self*)

Constructs a new component with the same parameters and random state.

Returns A new instance of this component with identical parameters and random state.

default_parameters(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

Returns Default parameters for this component.

Return type dict

describe(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

Parameters

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description 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

static load(*file_path*)

Loads component at file path.

Parameters **file_path** (*str*) – Location to load file.

Returns ComponentBase object

needs_fitting(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need 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 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*)

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.

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

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.
load	Loads component at file path.
needs_fitting	Returns boolean determining 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.

clone(*self*)

Constructs a new component with the same parameters and random state.

Returns A new instance of this component with identical parameters and random state.

default_parameters(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

Returns Default parameters for this component.

Return type dict

describe(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

Parameters

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description 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.

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 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

static load(*file_path*)

Loads component at file path.

Parameters **file_path** (*str*) – Location to load file.

Returns ComponentBase object

needs_fitting(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need 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 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*)

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.

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

hyperparameter_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.
load	Loads component at file path.
needs_fitting	Returns boolean determining 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.

clone(*self*)

Constructs a new component with the same parameters and random state.

Returns A new instance of this component with identical parameters and random state.

default_parameters(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

Returns Default parameters for this component.

Return type dict

describe(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

Parameters

- **print_name** (*bool, optional*) – whether to print name of component

- **return_dict** (*bool, optional*) – whether to return description 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*, *y=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

static load(*file_path*)

Loads component at file path.

Parameters **file_path** (*str*) – Location to load file.

Returns ComponentBase object

needs_fitting(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need 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 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*)

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.

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

hyperparameter_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.
load	Loads component at file path.
needs_fitting	Returns boolean determining 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.

clone(*self*)

Constructs a new component with the same parameters and random state.

Returns A new instance of this component with identical parameters and random state.

default_parameters(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

Returns Default parameters for this component.

Return type dict

describe(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

Parameters

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description 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.

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

static load(*file_path*)

Loads component at file path.

Parameters **file_path** (*str*) – Location to load file.

Returns ComponentBase object

needs_fitting(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need 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.

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.
load	Loads component at file path.
needs_fitting	Returns boolean determining 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.

clone(*self*)

Constructs a new component with the same parameters and random state.

Returns A new instance of this component with identical parameters and random state.

default_parameters(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

Returns Default parameters for this component.

Return type dict

describe(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

Parameters

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description 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*, *y=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

static load(*file_path*)

Loads component at file path.

Parameters **file_path** (*str*) – Location to load file.

Returns ComponentBase object

needs_fitting(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need 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 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*)

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.

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

hyperparameter_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.
load	Loads component at file path.
needs_fitting	Returns boolean determining 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.

clone(*self*)

Constructs a new component with the same parameters and random state.

Returns A new instance of this component with identical parameters and random state.

default_parameters(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

Returns Default parameters for this component.

Return type dict

describe(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

Parameters

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description 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.

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 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

static load(*file_path*)

Loads component at file path.

Parameters **file_path** (*str*) – Location to load file.

Returns ComponentBase object

needs_fitting(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need 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 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*)

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.

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

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.
load	Loads component at file path.
needs_fitting	Returns boolean determining 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.

clone(*self*)

Constructs a new component with the same parameters and random state.

Returns A new instance of this component with identical parameters and random state.

default_parameters(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

Returns Default parameters for this component.

Return type dict

describe(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

Parameters

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description 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*, *y=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

static load(*file_path*)

Loads component at file path.

Parameters **file_path** (*str*) – Location to load file.

Returns ComponentBase object

needs_fitting(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need 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 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*)

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.

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.
load	Loads component at file path.
needs_fitting	Returns boolean determining 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.

clone(*self*)

Constructs a new component with the same parameters and random state.

Returns A new instance of this component with identical parameters and random state.

default_parameters(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

Returns Default parameters for this component.

Return type dict

describe(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

Parameters

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description 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*, *y=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

static load(*file_path*)

Loads component at file path.

Parameters **file_path** (*str*) – Location to load file.

Returns ComponentBase object

needs_fitting(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need 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 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*)

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.

class evalml.pipelines.components.estimators.classifiers.**VowpalWabbitMulticlassClassifier**(*loss_function='lo learning_rate=0.5, decay_learning_rat 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

hyperparameter_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.
load	Loads component at file path.
needs_fitting	Returns boolean determining 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.

clone(*self*)

Constructs a new component with the same parameters and random state.

Returns A new instance of this component with identical parameters and random state.

default_parameters(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

Returns Default parameters for this component.

Return type dict

describe(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

Parameters

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description 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*, *y=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

static load(*file_path*)

Loads component at file path.

Parameters **file_path** (*str*) – Location to load file.

Returns ComponentBase object

needs_fitting(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need 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 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*)

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.

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

hyperparameter_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.
load	Loads component at file path.
needs_fitting	Returns boolean determining 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.

clone(*self*)

Constructs a new component with the same parameters and random state.

Returns A new instance of this component with identical parameters and random state.

default_parameters(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

Returns Default parameters for this component.

Return type dict

describe(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

Parameters

- **print_name** (*bool, optional*) – whether to print name of component

- **return_dict** (*bool, optional*) – whether to return description 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

static load(*file_path*)

Loads component at file path.

Parameters **file_path** (*str*) – Location to load file.

Returns ComponentBase object

needs_fitting(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need 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.

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=None*, *trend=None*, *start_p=2*, *d=0*, *start_q=2*, *max_p=5*, *max_d=2*, *max_q=5*, *seasonal=True*, *n_jobs=-1*, *random_seed=0*, *maxiter=10*, ***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.
- **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

hyperparameter_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],}
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.
load	Loads component at file path.
needs_fitting	Returns boolean determining 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.

clone(*self*)

Constructs a new component with the same parameters and random state.

Returns A new instance of this component with identical parameters and random state.

default_parameters(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

Returns Default parameters for this component.

Return type dict

describe(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

Parameters

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description 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 with a length of 1 as feature_importance is not defined for ARIMA regressor.

fit(*self*, *X*, *y=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 X was passed to *fit* but not passed in *predict*.

static load(*file_path*)

Loads component at file path.

Parameters **file_path** (*str*) – Location to load file.

Returns ComponentBase object

needs_fitting(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need 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*, *y=None*)

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*)

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.

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- parame- ter_ranges	{}
model_family	ModelFamily.BASELINE
modi- fies_features	True
modi- fies_target	False
name	Baseline Regressor
sup- ported_problem_types	[ProblemTypes.REGRESSION, ProblemTypes.TIME_SERIES_REGRESSION,]
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.
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.
load	Loads component at file path.
needs_fitting	Returns boolean determining 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.

clone(*self*)

Constructs a new component with the same parameters and random state.

Returns A new instance of this component with identical parameters and random state.

default_parameters(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

Returns Default parameters for this component.

Return type dict

describe(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

Parameters

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description 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.

static load(*file_path*)

Loads component at file path.

Parameters **file_path** (*str*) – Location to load file.

Returns ComponentBase object

needs_fitting(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need 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*)

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.

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 bootstrap_type=N silent=False, allow_writing_ 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

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 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.
load	Loads component at file path.
needs_fitting	Returns boolean determining 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.

clone(*self*)

Constructs a new component with the same parameters and random state.

Returns A new instance of this component with identical parameters and random state.

default_parameters(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

Returns Default parameters for this component.

Return type dict

describe(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

Parameters

- **print_name** (*bool, optional*) – whether to print name of component

- **return_dict** (*bool, optional*) – whether to return description 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

static load(*file_path*)

Loads component at file path.

Parameters **file_path** (*str*) – Location to load file.

Returns ComponentBase object

needs_fitting(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need 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 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*)

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.

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** (*{"mse", "friedman_mse", "mae", "poisson"}*) – The function to measure the quality of a split. Supported criteria are:
  - "mse" 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
  - "mae" 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": ["mse", "friedman_mse", "mae"], "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.
load	Loads component at file path.
needs_fitting	Returns boolean determining 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.

clone(*self*)

Constructs a new component with the same parameters and random state.

Returns A new instance of this component with identical parameters and random state.

default_parameters(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

Returns Default parameters for this component.

Return type dict

describe(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

Parameters

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description 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.

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 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

static load(*file_path*)

Loads component at file path.

Parameters **file_path** (*str*) – Location to load file.

Returns ComponentBase object

needs_fitting(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need 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 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*)

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.

elasticnet_regressor

Elastic Net Regressor.

Module Contents

Classes Summary

ElasticNetRegressor	Elastic Net Regressor.

Contents

class evalml.pipelines.components.estimators.regressors.elasticnet_regressor.**ElasticNetRegressor**(*alpha=0.0001, l1_ratio=0.15, max_iter=1000, normalize=False, 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.
- **normalize** (*boolean*) – If True, the regressors will be normalized before regression by subtracting the mean and dividing by the l2-norm. Defaults to False.
- **random_seed** (*int*) – Seed for the random number generator. Defaults to 0.

Attributes

hyperparameter_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.
load	Loads component at file path.
needs_fitting	Returns boolean determining 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.

clone(*self*)

Constructs a new component with the same parameters and random state.

Returns A new instance of this component with identical parameters and random state.

default_parameters(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

Returns Default parameters for this component.

Return type dict

describe(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

Parameters

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description 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*, *y=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

static load(*file_path*)

Loads component at file path.

Parameters **file_path** (*str*) – Location to load file.

Returns ComponentBase object

needs_fitting(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need 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 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*)

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.

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=100*, *max_features='aut* *max_depth=6*, *min_samples_split* *min_weight_fracti* *n_jobs=-1*, *random_seed=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

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 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.
load	Loads component at file path.
needs_fitting	Returns boolean determining 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.

clone(*self*)

Constructs a new component with the same parameters and random state.

Returns A new instance of this component with identical parameters and random state.

default_parameters(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

Returns Default parameters for this component.

Return type dict

describe(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

Parameters

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description 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.

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 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

static load(*file_path*)

Loads component at file path.

Parameters **file_path** (*str*) – Location to load file.

Returns ComponentBase object

needs_fitting(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need 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 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*)

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.

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

hyperparameter_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.
load	Loads component at file path.
needs_fitting	Returns boolean determining 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.

clone(*self*)

Constructs a new component with the same parameters and random state.

Returns A new instance of this component with identical parameters and random state.

default_parameters(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

Returns Default parameters for this component.

Return type dict

describe(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

Parameters

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description 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 with a length of 1 as feature_importance is not defined for Exponential Smoothing regressor.

fit(*self*, *X*, *y=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.

static load(*file_path*)

Loads component at file path.

Parameters **file_path** (*str*) – Location to load file.

Returns ComponentBase object

needs_fitting(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need 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*, *y=None*)

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*)

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.

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=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 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

hyperparameter_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.
load	Loads component at file path.
needs_fitting	Returns boolean determining 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.

clone(*self*)

Constructs a new component with the same parameters and random state.

Returns A new instance of this component with identical parameters and random state.

default_parameters(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

Returns Default parameters for this component.

Return type dict

describe(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

Parameters

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description 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.

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

static load(*file_path*)

Loads component at file path.

Parameters **file_path** (*str*) – Location to load file.

Returns ComponentBase object

needs_fitting(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need 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*)

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.

linear_regressor

Linear Regressor.

Module Contents

Classes Summary

LinearRegressor	Linear Regressor.

Contents

class evalml.pipelines.components.estimators.regressors.linear_regressor.**LinearRegressor**(*fit_intercept=True*, *normalize=False*, *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.
- **normalize** (*boolean*) – If True, the regressors will be normalized before regression by subtracting the mean and dividing by the l2-norm. This parameter is ignored when fit_intercept is set to False. Defaults to False.

- **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], "normalize": [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.
load	Loads component at file path.
needs_fitting	Returns boolean determining 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.

clone(*self*)

Constructs a new component with the same parameters and random state.

Returns A new instance of this component with identical parameters and random state.

default_parameters(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

Returns Default parameters for this component.

Return type dict

describe(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

Parameters

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description 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*, *y=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

static load(*file_path*)

Loads component at file path.

Parameters **file_path** (*str*) – Location to load file.

Returns ComponentBase object

needs_fitting(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need 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 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*)

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.

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=Nor*
change-
point_prior_sca
sea-
son-
al-
ity_prior_scale=
hol-
i-
days_prior_scal
sea-
son-
al-
ity_mode='addi
ran-
dom_seed=0,
stan_backend='
***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/

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.
load	Loads component at file path.
needs_fitting	Returns boolean determining 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.

static build_prophet_df(*X*, *y=None*, *time_index='ds'*)

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*)

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*)

Returns array of 0's with len(1) as feature_importance is not defined for Prophet regressor.

fit(*self*, *X*, *y=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*)

Get parameters for the Prophet regressor.

static load(*file_path*)

Loads component at file path.

Parameters **file_path** (*str*) – Location to load file.

Returns ComponentBase object

needs_fitting(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need 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*, *y=None*)

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*)

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.

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=100*, *max_depth=6*, *n_jobs=-1*, *random_seed=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.
load	Loads component at file path.
needs_fitting	Returns boolean determining 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.

clone(*self*)

Constructs a new component with the same parameters and random state.

Returns A new instance of this component with identical parameters and random state.

default_parameters(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

Returns Default parameters for this component.

Return type dict

describe(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

Parameters

- **print_name** (*bool, optional*) – whether to print name of component

- **return_dict** (*bool, optional*) – whether to return description 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.

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 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

static load(*file_path*)

Loads component at file path.

Parameters **file_path** (*str*) – Location to load file.

Returns ComponentBase object

needs_fitting(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need 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 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*)

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.

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

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.
load	Loads component at file path.
needs_fitting	Returns boolean determining 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.

clone(*self*)

Constructs a new component with the same parameters and random state.

Returns A new instance of this component with identical parameters and random state.

default_parameters(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

Returns Default parameters for this component.

Return type dict

describe(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

Parameters

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description 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*, *y=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

static load(*file_path*)

Loads component at file path.

Parameters **file_path** (*str*) – Location to load file.

Returns ComponentBase object

needs_fitting(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need 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 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*)

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.

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.
load	Loads component at file path.
needs_fitting	Returns boolean determining 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.

clone(*self*)

Constructs a new component with the same parameters and random state.

Returns A new instance of this component with identical parameters and random state.

default_parameters(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

Returns Default parameters for this component.

Return type dict

describe(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

Parameters

- **print_name** (*bool, optional*) – whether to print name of component

- **return_dict** (*bool, optional*) – whether to return description 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.

static load(*file_path*)

Loads component at file path.

Parameters **file_path** (*str*) – Location to load file.

Returns ComponentBase object

needs_fitting(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need 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.

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

hyperparameter_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.
load	Loads component at file path.
needs_fitting	Returns boolean determining 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.

clone(*self*)

Constructs a new component with the same parameters and random state.

Returns A new instance of this component with identical parameters and random state.

default_parameters(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

Returns Default parameters for this component.

Return type dict

describe(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

Parameters

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description 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*, *y=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

static load(*file_path*)

Loads component at file path.

Parameters **file_path** (*str*) – Location to load file.

Returns ComponentBase object

needs_fitting(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need 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 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*)

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.

xgboost_regressor

XGBoost Regressor.

Module Contents

Classes Summary

XGBoostRegressor	XGBoost Regressor.

Contents

class evalml.pipelines.components.estimators.regressors.xgboost_regressor.**XGBoostRegressor**(*eta=0.1*, *max_depth=6*, *min_child_weig*
n_estimators=1
ran-
dom_seed=0, *n_jobs=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

hyperparameter_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.
load	Loads component at file path.
needs_fitting	Returns boolean determining 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.

clone(*self*)

Constructs a new component with the same parameters and random state.

Returns A new instance of this component with identical parameters and random state.

default_parameters(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

Returns Default parameters for this component.

Return type dict

describe(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

Parameters

- **print_name** (*bool, optional*) – whether to print name of component

- **return_dict** (*bool, optional*) – whether to return description 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 regressor.

fit(*self*, *X*, *y=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

static load(*file_path*)

Loads component at file path.

Parameters **file_path** (*str*) – Location to load file.

Returns ComponentBase object

needs_fitting(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need 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 XGBoost regressor.

Parameters **X** (*pd.DataFrame*) – Data of shape [n_samples, n_features].

Returns Predicted values.

Return type pd.Series

predict_proba(*self*, *X*)

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.

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=None*, *trend=None*, *start_p=2*, *d=0*, *start_q=2*, *max_p=5*, *max_d=2*, *max_q=5*, *seasonal=True*, *n_jobs=- 1*, *random_seed=0*, *maxiter=10*, ***kwargs*)

Autoregressive Integrated Moving Average Model. The three parameters (p, d, q) are the AR order, the de-

gree 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.
- **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

hyperparameter_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],}
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.
load	Loads component at file path.
needs_fitting	Returns boolean determining 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.

clone(*self*)

Constructs a new component with the same parameters and random state.

Returns A new instance of this component with identical parameters and random state.

default_parameters(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

Returns Default parameters for this component.

Return type dict

describe(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

Parameters

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description 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 with a length of 1 as feature_importance is not defined for ARIMA regressor.

fit(*self*, *X*, *y=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 X was passed to *fit* but not passed in *predict*.

static load(*file_path*)

Loads component at file path.

Parameters **file_path** (*str*) – Location to load file.

Returns ComponentBase object

needs_fitting(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need 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*, *y=None*)

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*)

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.

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

hyperparameter_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.
load	Loads component at file path.
needs_fitting	Returns boolean determining 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.

clone(*self*)

Constructs a new component with the same parameters and random state.

Returns A new instance of this component with identical parameters and random state.

default_parameters(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

Returns Default parameters for this component.

Return type dict

describe(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

Parameters

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description 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.

static load(*file_path*)

Loads component at file path.

Parameters **file_path** (*str*) – Location to load file.

Returns ComponentBase object

needs_fitting(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need 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*)

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.

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

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 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.
load	Loads component at file path.
needs_fitting	Returns boolean determining 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.

clone(*self*)

Constructs a new component with the same parameters and random state.

Returns A new instance of this component with identical parameters and random state.

default_parameters(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

Returns Default parameters for this component.

Return type dict

describe(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

Parameters

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description 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

static load(*file_path*)

Loads component at file path.

Parameters **file_path** (*str*) – Location to load file.

Returns ComponentBase object

needs_fitting(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need 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 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*)

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.

class evalml.pipelines.components.estimators.regressors.**DecisionTreeRegressor**(*criterion='mse'*, *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** (*{"mse", "friedman_mse", "mae", "poisson"}*) – The function to measure the quality of a split. Supported criteria are:
  - "mse" 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
  - "mae" 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": ["mse", "friedman_mse", "mae"], "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.
load	Loads component at file path.
needs_fitting	Returns boolean determining 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.

clone(*self*)

Constructs a new component with the same parameters and random state.

Returns A new instance of this component with identical parameters and random state.

default_parameters(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

Returns Default parameters for this component.

Return type dict

describe(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

Parameters

- **print_name** (*bool, optional*) – whether to print name of component

- **return_dict** (*bool, optional*) – whether to return description 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.

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 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

static load(*file_path*)

Loads component at file path.

Parameters **file_path** (*str*) – Location to load file.

Returns ComponentBase object

needs_fitting(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need 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 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*)

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.

class evalml.pipelines.components.estimators.regressors.**ElasticNetRegressor**(*alpha=0.0001*, *l1_ratio=0.15*, *max_iter=1000*, *normalize=False*, *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.
- **normalize** (*boolean*) – If True, the regressors will be normalized before regression by subtracting the mean and dividing by the l2-norm. Defaults to False.
- **random_seed** (*int*) – Seed for the random number generator. Defaults to 0.

Attributes

hyperparameter_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.
load	Loads component at file path.
needs_fitting	Returns boolean determining 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.

clone(*self*)

Constructs a new component with the same parameters and random state.

Returns A new instance of this component with identical parameters and random state.

default_parameters(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

Returns Default parameters for this component.

Return type dict

describe(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

Parameters

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description 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*, *y=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

static load(*file_path*)

Loads component at file path.

Parameters **file_path** (*str*) – Location to load file.

Returns ComponentBase object

needs_fitting(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need 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 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*)

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.

class evalml.pipelines.components.estimators.regressors.**ExponentialSmoothingRegressor**(*trend=None*, *damped_trend=False*, *seasonal=None*, *sp=2*, *n_jobs=-1*, *random_seed=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

hyperparameter_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.
load	Loads component at file path.
needs_fitting	Returns boolean determining 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.

clone(*self*)

Constructs a new component with the same parameters and random state.

Returns A new instance of this component with identical parameters and random state.

default_parameters(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

Returns Default parameters for this component.

Return type dict

describe(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

Parameters

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description 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 with a length of 1 as feature_importance is not defined for Exponential Smoothing regressor.

fit(*self*, *X*, *y=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.

static load(*file_path*)

Loads component at file path.

Parameters **file_path** (*str*) – Location to load file.

Returns ComponentBase object

needs_fitting(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need 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*, *y=None*)

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*)

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.

class evalml.pipelines.components.estimators.regressors.**ExtraTreesRegressor**(*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 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

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 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.
load	Loads component at file path.
needs_fitting	Returns boolean determining 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.

clone(*self*)

Constructs a new component with the same parameters and random state.

Returns A new instance of this component with identical parameters and random state.

default_parameters(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

Returns Default parameters for this component.

Return type dict

describe(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

Parameters

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description 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.

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 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

static load(*file_path*)

Loads component at file path.

Parameters **file_path** (*str*) – Location to load file.

Returns ComponentBase object

needs_fitting(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need 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 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*)

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.

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

hyperparameter_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.
load	Loads component at file path.
needs_fitting	Returns boolean determining 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.

clone(*self*)

Constructs a new component with the same parameters and random state.

Returns A new instance of this component with identical parameters and random state.

default_parameters(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

Returns Default parameters for this component.

Return type dict

describe(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

Parameters

- **print_name** (*bool, optional*) – whether to print name of component

- **return_dict** (*bool, optional*) – whether to return description 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.

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

static load(*file_path*)

Loads component at file path.

Parameters **file_path** (*str*) – Location to load file.

Returns ComponentBase object

needs_fitting(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need 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*)

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.

class evalml.pipelines.components.estimators.regressors.**LinearRegressor**(*fit_intercept=True*, *normalize=False*, *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.
- **normalize** (*boolean*) – If True, the regressors will be normalized before regression by subtracting the mean and dividing by the l2-norm. This parameter is ignored when fit_intercept is set to False. Defaults to False.
- **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], "normalize": [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.
load	Loads component at file path.
needs_fitting	Returns boolean determining 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.

clone(*self*)

Constructs a new component with the same parameters and random state.

Returns A new instance of this component with identical parameters and random state.

default_parameters(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

Returns Default parameters for this component.

Return type dict

describe(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

Parameters

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description 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*, *y=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

static load(*file_path*)

Loads component at file path.

Parameters **file_path** (*str*) – Location to load file.

Returns ComponentBase object

needs_fitting(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need 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 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*)

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.

class evalml.pipelines.components.estimators.regressors.**ProphetRegressor**(*time_index=None*, *changepoint_prior_scale=0.05*, *seasonality_prior_scale=10*, *holidays_prior_scale=10*, *seasonality_mode='additive'*, *random_seed=0*, *stan_backend='CMDSTANPY'*, ***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/

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.
load	Loads component at file path.
needs_fitting	Returns boolean determining 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.

static build_prophet_df(*X*, *y=None*, *time_index='ds'*)

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*)

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*)

Returns array of 0's with len(1) as feature_importance is not defined for Prophet regressor.

fit(*self*, *X*, *y=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*)

Get parameters for the Prophet regressor.

static load(*file_path*)

Loads component at file path.

Parameters **file_path** (*str*) – Location to load file.

Returns ComponentBase object

needs_fitting(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need 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*, *y=None*)

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*)

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.

class evalml.pipelines.components.estimators.regressors.**RandomForestRegressor**(*n_estimators=100*, *max_depth=6*, *n_jobs=- 1*, *random_seed=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.
load	Loads component at file path.
needs_fitting	Returns boolean determining 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.

clone(*self*)

Constructs a new component with the same parameters and random state.

Returns A new instance of this component with identical parameters and random state.

default_parameters(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

Returns Default parameters for this component.

Return type dict

describe(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

Parameters

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description 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.

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 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

static load(*file_path*)

Loads component at file path.

Parameters **file_path** (*str*) – Location to load file.

Returns ComponentBase object

needs_fitting(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need 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 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*)

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.

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

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.
load	Loads component at file path.
needs_fitting	Returns boolean determining 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.

clone(*self*)

Constructs a new component with the same parameters and random state.

Returns A new instance of this component with identical parameters and random state.

default_parameters(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

Returns Default parameters for this component.

Return type dict

describe(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

Parameters

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description 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*, *y=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

static load(*file_path*)

Loads component at file path.

Parameters **file_path** (*str*) – Location to load file.

Returns ComponentBase object

needs_fitting(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need 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 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*)

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.

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

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.
load	Loads component at file path.
needs_fitting	Returns boolean determining 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.

clone(*self*)

Constructs a new component with the same parameters and random state.

Returns A new instance of this component with identical parameters and random state.

default_parameters(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

Returns Default parameters for this component.

Return type dict

describe(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

Parameters

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description 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.

static load(*file_path*)

Loads component at file path.

Parameters **file_path** (*str*) – Location to load file.

Returns ComponentBase object

needs_fitting(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need 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.

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

hyperparameter_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.
load	Loads component at file path.
needs_fitting	Returns boolean determining 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.

clone(*self*)

Constructs a new component with the same parameters and random state.

Returns A new instance of this component with identical parameters and random state.

default_parameters(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

Returns Default parameters for this component.

Return type dict

describe(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

Parameters

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description 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*, *y=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

static load(*file_path*)

Loads component at file path.

Parameters **file_path** (*str*) – Location to load file.

Returns ComponentBase object

needs_fitting(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need 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 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*)

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.

class evalml.pipelines.components.estimators.regressors.**XGBoostRegressor**(*eta=0.1*, *max_depth=6*, *min_child_weight=1*, *n_estimators=100*, *random_seed=0*, *n_jobs=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

hyperparameter_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.
load	Loads component at file path.
needs_fitting	Returns boolean determining 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.

clone(*self*)

Constructs a new component with the same parameters and random state.

Returns A new instance of this component with identical parameters and random state.

default_parameters(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

Returns Default parameters for this component.

Return type dict

describe(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

Parameters

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description 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 regressor.

fit(*self*, *X*, *y=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

static load(*file_path*)

Loads component at file path.

Parameters **file_path** (*str*) – Location to load file.

Returns ComponentBase object

needs_fitting(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need 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 XGBoost regressor.

Parameters **X** (*pd.DataFrame*) – Data of shape [n_samples, n_features].

Returns Predicted values.

Return type pd.Series

predict_proba(*self*, *X*)

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.

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=None*, *component_obj=None*, *random_seed=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.
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.

clone(*self*)

Constructs a new component with the same parameters and random state.

Returns A new instance of this component with identical parameters and random state.

default_parameters(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

Returns Default parameters for this component.

Return type dict

describe(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

Parameters

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description 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.

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 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

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*)

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*)

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.

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=None*, *trend=None*, *start_p=2*, *d=0*, *start_q=2*, *max_p=5*, *max_d=2*, *max_q=5*, *seasonal=True*, *n_jobs=- 1*, *random_seed=0*, *maxiter=10*, ***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.
- **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

hyperparameter_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],}
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.
load	Loads component at file path.
needs_fitting	Returns boolean determining 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.

clone(*self*)

Constructs a new component with the same parameters and random state.

Returns A new instance of this component with identical parameters and random state.

default_parameters(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

Returns Default parameters for this component.

Return type dict

describe(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

Parameters

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description 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 with a length of 1 as feature_importance is not defined for ARIMA regressor.

fit(*self*, *X*, *y=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 X was passed to *fit* but not passed in *predict*.

static load(*file_path*)

Loads component at file path.

Parameters **file_path** (*str*) – Location to load file.

Returns ComponentBase object

needs_fitting(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need 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*, *y=None*)

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*)

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.

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

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.
load	Loads component at file path.
needs_fitting	Returns boolean determining 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.

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.

static load(*file_path*)

Loads component at file path.

Parameters **file_path** (*str*) – Location to load file.

Returns ComponentBase object

needs_fitting(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need 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.

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

hyperparameter_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.
load	Loads component at file path.
needs_fitting	Returns boolean determining 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.

clone(*self*)

Constructs a new component with the same parameters and random state.

Returns A new instance of this component with identical parameters and random state.

default_parameters(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

Returns Default parameters for this component.

Return type dict

describe(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

Parameters

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description 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.

static load(*file_path*)

Loads component at file path.

Parameters **file_path** (*str*) – Location to load file.

Returns ComponentBase object

needs_fitting(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need 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*)

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.

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.
load	Loads component at file path.
needs_fitting	Returns boolean determining 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 probability estimates for labels.
save	Saves component at file path.

clone(*self*)

Constructs a new component with the same parameters and random state.

Returns A new instance of this component with identical parameters and random state.

default_parameters(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

Returns Default parameters for this component.

Return type dict

describe(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

Parameters

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description 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

static load(*file_path*)

Loads component at file path.

Parameters **file_path** (*str*) – Location to load file.

Returns ComponentBase object

needs_fitting(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need 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.DataFrame

predict_proba(*self*, *X*)

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.

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

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 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.
load	Loads component at file path.
needs_fitting	Returns boolean determining 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.

clone(*self*)

Constructs a new component with the same parameters and random state.

Returns A new instance of this component with identical parameters and random state.

default_parameters(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

Returns Default parameters for this component.

Return type dict

describe(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

Parameters

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description 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

static load(*file_path*)

Loads component at file path.

Parameters **file_path** (*str*) – Location to load file.

Returns ComponentBase object

needs_fitting(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need 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 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*)

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.

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

hyperparameter_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.
load	Loads component at file path.
needs_fitting	Returns boolean determining 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.

clone(*self*)

Constructs a new component with the same parameters and random state.

Returns A new instance of this component with identical parameters and random state.

default_parameters(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

Returns Default parameters for this component.

Return type dict

describe(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

Parameters

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description 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.

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 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

static load(*file_path*)

Loads component at file path.

Parameters **file_path** (*str*) – Location to load file.

Returns ComponentBase object

needs_fitting(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need 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 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*)

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.

class evalml.pipelines.components.estimators.**DecisionTreeRegressor**(*criterion='mse'*, *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** (*{"mse", "friedman_mse", "mae", "poisson"}*) – The function to measure the quality of a split. Supported criteria are:
  - "mse" 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
  - "mae" 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": ["mse", "friedman_mse", "mae"], "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.
load	Loads component at file path.
needs_fitting	Returns boolean determining 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.

clone(*self*)

Constructs a new component with the same parameters and random state.

Returns A new instance of this component with identical parameters and random state.

default_parameters(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

Returns Default parameters for this component.

Return type dict

describe(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

Parameters

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description 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.

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 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

static load(*file_path*)

Loads component at file path.

Parameters **file_path** (*str*) – Location to load file.

Returns ComponentBase object

needs_fitting(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need 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 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*)

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.

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

hyperparameter_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.
load	Loads component at file path.
needs_fitting	Returns boolean determining 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.

clone(*self*)

Constructs a new component with the same parameters and random state.

Returns A new instance of this component with identical parameters and random state.

default_parameters(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

Returns Default parameters for this component.

Return type dict

`describe`(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

Parameters

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description 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

`static load`(*file_path*)

Loads component at file path.

Parameters **file_path** (*str*) – Location to load file.

Returns ComponentBase object

`needs_fitting`(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need 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 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*)

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.

class evalml.pipelines.components.estimators.**ElasticNetRegressor**(*alpha=0.0001*, *l1_ratio=0.15*, *max_iter=1000*, *normalize=False*, *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.
- **normalize** (*boolean*) – If True, the regressors will be normalized before regression by subtracting the mean and dividing by the l2-norm. Defaults to False.
- **random_seed** (*int*) – Seed for the random number generator. Defaults to 0.

Attributes

hyperparameter_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.
load	Loads component at file path.
needs_fitting	Returns boolean determining 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.

clone(*self*)

Constructs a new component with the same parameters and random state.

Returns A new instance of this component with identical parameters and random state.

default_parameters(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

Returns Default parameters for this component.

Return type dict

describe(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

Parameters

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description 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*, *y=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

static load(*file_path*)

Loads component at file path.

Parameters **file_path** (*str*) – Location to load file.

Returns ComponentBase object

needs_fitting(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need 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 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*)

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.

class evalml.pipelines.components.estimators.**Estimator**(*parameters=None*, *component_obj=None*, *random_seed=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.
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.

clone(*self*)

Constructs a new component with the same parameters and random state.

Returns A new instance of this component with identical parameters and random state.

default_parameters(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

Returns Default parameters for this component.

Return type dict

describe(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

Parameters

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description 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.

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 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

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*)

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*)

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.

class evalml.pipelines.components.estimators.**ExponentialSmoothingRegressor**(*trend=None*, *damped_trend=False*, *seasonal=None*, *sp=2*, *n_jobs=- 1*, *random_seed=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

hyperparameter_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.
load	Loads component at file path.
needs_fitting	Returns boolean determining 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.

clone(*self*)

Constructs a new component with the same parameters and random state.

Returns A new instance of this component with identical parameters and random state.

default_parameters(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

Returns Default parameters for this component.

Return type dict

describe(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

Parameters

- **print_name** (*bool, optional*) – whether to print name of component

- **return_dict** (*bool, optional*) – whether to return description 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 with a length of 1 as feature_importance is not defined for Exponential Smoothing regressor.

fit(*self*, *X*, *y=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.

static load(*file_path*)

Loads component at file path.

Parameters **file_path** (*str*) – Location to load file.

Returns ComponentBase object

needs_fitting(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need 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*, *y=None*)

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*)

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.

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.
load	Loads component at file path.
needs_fitting	Returns boolean determining 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.

`clone`(*self*)

Constructs a new component with the same parameters and random state.

Returns A new instance of this component with identical parameters and random state.

`default_parameters`(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

Returns Default parameters for this component.

Return type dict

`describe`(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

Parameters

- **print_name** (*bool, optional*) – whether to print name of component

- **return_dict** (*bool, optional*) – whether to return description 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.

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 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

static load(*file_path*)

Loads component at file path.

Parameters **file_path** (*str*) – Location to load file.

Returns ComponentBase object

needs_fitting(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need 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 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*)

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.

class evalml.pipelines.components.estimators.**ExtraTreesRegressor**(*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 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

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 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.
load	Loads component at file path.
needs_fitting	Returns boolean determining 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.

clone(*self*)

Constructs a new component with the same parameters and random state.

Returns A new instance of this component with identical parameters and random state.

default_parameters(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

Returns Default parameters for this component.

Return type dict

describe(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

Parameters

- **print_name** (*bool, optional*) – whether to print name of component

- **return_dict** (*bool, optional*) – whether to return description 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.

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 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

static load(*file_path*)

Loads component at file path.

Parameters **file_path** (*str*) – Location to load file.

Returns ComponentBase object

needs_fitting(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need 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 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*)

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.

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

hyperparameter_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.
load	Loads component at file path.
needs_fitting	Returns boolean determining 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.

clone(*self*)

Constructs a new component with the same parameters and random state.

Returns A new instance of this component with identical parameters and random state.

default_parameters(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

Returns Default parameters for this component.

Return type dict

describe(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

Parameters

- **print_name** (*bool, optional*) – whether to print name of component

- **return_dict** (*bool, optional*) – whether to return description 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*, *y=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

static load(*file_path*)

Loads component at file path.

Parameters **file_path** (*str*) – Location to load file.

Returns ComponentBase object

needs_fitting(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need 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 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*)

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.

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

hyperparameter_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.
load	Loads component at file path.
needs_fitting	Returns boolean determining 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.

clone(*self*)

Constructs a new component with the same parameters and random state.

Returns A new instance of this component with identical parameters and random state.

default_parameters(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

Returns Default parameters for this component.

Return type dict

describe(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

Parameters

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description 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.

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

static load(*file_path*)

Loads component at file path.

Parameters **file_path** (*str*) – Location to load file.

Returns ComponentBase object

needs_fitting(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need 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.

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

hyperparameter_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.
load	Loads component at file path.
needs_fitting	Returns boolean determining 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.

clone(*self*)

Constructs a new component with the same parameters and random state.

Returns A new instance of this component with identical parameters and random state.

default_parameters(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

Returns Default parameters for this component.

Return type dict

describe(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

Parameters

- **print_name** (*bool, optional*) – whether to print name of component

- **return_dict** (*bool, optional*) – whether to return description 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.

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

static load(*file_path*)

Loads component at file path.

Parameters **file_path** (*str*) – Location to load file.

Returns ComponentBase object

needs_fitting(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need 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*)

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.

class evalml.pipelines.components.estimators.**LinearRegressor**(*fit_intercept=True*, *normalize=False*, *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.
- **normalize** (*boolean*) – If True, the regressors will be normalized before regression by subtracting the mean and dividing by the l2-norm. This parameter is ignored when fit_intercept is set to False. Defaults to False.
- **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], “normalize”: [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.
load	Loads component at file path.
needs_fitting	Returns boolean determining 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.

clone(*self*)

Constructs a new component with the same parameters and random state.

Returns A new instance of this component with identical parameters and random state.

default_parameters(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

Returns Default parameters for this component.

Return type dict

describe(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

Parameters

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description 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*, *y=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

static load(*file_path*)

Loads component at file path.

Parameters **file_path** (*str*) – Location to load file.

Returns ComponentBase object

needs_fitting(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need 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 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*)

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.

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

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.
load	Loads component at file path.
needs_fitting	Returns boolean determining 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.

clone(*self*)

Constructs a new component with the same parameters and random state.

Returns A new instance of this component with identical parameters and random state.

default_parameters(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

Returns Default parameters for this component.

Return type dict

describe(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

Parameters

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description 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*, *y=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

static load(*file_path*)

Loads component at file path.

Parameters **file_path** (*str*) – Location to load file.

Returns ComponentBase object

needs_fitting(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need 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 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*)

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.

class evalml.pipelines.components.estimators.**ProphetRegressor**(*time_index=None*, *changepoint_prior_scale=0.05*, *seasonality_prior_scale=10*, *holidays_prior_scale=10*, *seasonality_mode='additive'*, *random_seed=0*, *stan_backend='CMDSTANPY'*, ***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/

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.
load	Loads component at file path.
needs_fitting	Returns boolean determining 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.

static build_prophet_df(*X*, *y=None*, *time_index='ds'*)

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*)

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*)

Returns array of 0's with len(1) as feature_importance is not defined for Prophet regressor.

fit(*self*, *X*, *y=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*)

Get parameters for the Prophet regressor.

static load(*file_path*)

Loads component at file path.

Parameters **file_path** (*str*) – Location to load file.

Returns ComponentBase object

needs_fitting(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need 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*, *y=None*)

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*)

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.

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

hyperparameter_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.
load	Loads component at file path.
needs_fitting	Returns boolean determining 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.

clone(*self*)

Constructs a new component with the same parameters and random state.

Returns A new instance of this component with identical parameters and random state.

default_parameters(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

Returns Default parameters for this component.

Return type dict

describe(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

Parameters

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description 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.

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 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

static load(*file_path*)

Loads component at file path.

Parameters **file_path** (*str*) – Location to load file.

Returns ComponentBase object

needs_fitting(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need 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 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*)

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.

class evalml.pipelines.components.estimators.**RandomForestRegressor**(*n_estimators=100*, *max_depth=6*, *n_jobs=- 1*, *random_seed=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.
load	Loads component at file path.
needs_fitting	Returns boolean determining 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.

clone(*self*)

Constructs a new component with the same parameters and random state.

Returns A new instance of this component with identical parameters and random state.

default_parameters(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

Returns Default parameters for this component.

Return type dict

describe(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

Parameters

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description 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.

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 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

static load(*file_path*)

Loads component at file path.

Parameters **file_path** (*str*) – Location to load file.

Returns ComponentBase object

needs_fitting(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need 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 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*)

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.

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

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.
load	Loads component at file path.
needs_fitting	Returns boolean determining 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.

clone(*self*)

Constructs a new component with the same parameters and random state.

Returns A new instance of this component with identical parameters and random state.

default_parameters(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

Returns Default parameters for this component.

Return type dict

describe(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

Parameters

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description 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*, *y=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

static load(*file_path*)

Loads component at file path.

Parameters **file_path** (*str*) – Location to load file.

Returns ComponentBase object

needs_fitting(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need 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 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*)

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.

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.
load	Loads component at file path.
needs_fitting	Returns boolean determining 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.

clone(*self*)

Constructs a new component with the same parameters and random state.

Returns A new instance of this component with identical parameters and random state.

default_parameters(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

Returns Default parameters for this component.

Return type dict

describe(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

Parameters

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description 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*, *y=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

static load(*file_path*)

Loads component at file path.

Parameters **file_path** (*str*) – Location to load file.

Returns ComponentBase object

needs_fitting(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need 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 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*)

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.

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

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.
load	Loads component at file path.
needs_fitting	Returns boolean determining 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.

clone(*self*)

Constructs a new component with the same parameters and random state.

Returns A new instance of this component with identical parameters and random state.

default_parameters(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

Returns Default parameters for this component.

Return type dict

describe(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

Parameters

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description 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.

static load(*file_path*)

Loads component at file path.

Parameters **file_path** (*str*) – Location to load file.

Returns ComponentBase object

needs_fitting(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need 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.

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

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.
load	Loads component at file path.
needs_fitting	Returns boolean determining 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.

clone(*self*)

Constructs a new component with the same parameters and random state.

Returns A new instance of this component with identical parameters and random state.

default_parameters(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

Returns Default parameters for this component.

Return type dict

describe(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

Parameters

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description 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*, *y=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

static load(*file_path*)

Loads component at file path.

Parameters **file_path** (*str*) – Location to load file.

Returns ComponentBase object

needs_fitting(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need 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 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*)

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.

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

hyperparameter_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.
load	Loads component at file path.
needs_fitting	Returns boolean determining 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.

clone(*self*)

Constructs a new component with the same parameters and random state.

Returns A new instance of this component with identical parameters and random state.

default_parameters(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

Returns Default parameters for this component.

Return type dict

describe(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

Parameters

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description 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*, *y=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

static load(*file_path*)

Loads component at file path.

Parameters **file_path** (*str*) – Location to load file.

Returns ComponentBase object

needs_fitting(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need 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 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*)

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.

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

hyperparameter_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.
load	Loads component at file path.
needs_fitting	Returns boolean determining 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.

clone(*self*)

Constructs a new component with the same parameters and random state.

Returns A new instance of this component with identical parameters and random state.

default_parameters(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

Returns Default parameters for this component.

Return type dict

describe(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

Parameters

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description 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*, *y=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

static load(*file_path*)

Loads component at file path.

Parameters **file_path** (*str*) – Location to load file.

Returns ComponentBase object

needs_fitting(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need 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 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*)

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.

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

hyperparameter_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.
load	Loads component at file path.
needs_fitting	Returns boolean determining 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.

clone(*self*)

Constructs a new component with the same parameters and random state.

Returns A new instance of this component with identical parameters and random state.

default_parameters(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

Returns Default parameters for this component.

Return type dict

describe(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

Parameters

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description 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

static load(*file_path*)

Loads component at file path.

Parameters **file_path** (*str*) – Location to load file.

Returns ComponentBase object

needs_fitting(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need 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.

class evalml.pipelines.components.estimators.**XGBoostRegressor**(*eta=0.1*, *max_depth=6*, *min_child_weight=1*, *n_estimators=100*, *random_seed=0*, *n_jobs=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

hyperparameter_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.
load	Loads component at file path.
needs_fitting	Returns boolean determining 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.

clone(*self*)

Constructs a new component with the same parameters and random state.

Returns A new instance of this component with identical parameters and random state.

default_parameters(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

Returns Default parameters for this component.

Return type dict

describe(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

Parameters

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description 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 regressor.

fit(*self*, *X*, *y=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

static load(*file_path*)

Loads component at file path.

Parameters **file_path** (*str*) – Location to load file.

Returns ComponentBase object

needs_fitting(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need 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 XGBoost regressor.

Parameters **X** (*pd.DataFrame*) – Data of shape [n_samples, n_features].

Returns Predicted values.

Return type pd.Series

predict_proba(*self*, *X*)

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.

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

hyperparameter_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.

clone(*self*)

Constructs a new component with the same parameters and random state.

Returns A new instance of this component with identical parameters and random state.

default_parameters(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

Returns Default parameters for this component.

Return type dict

describe(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

Parameters

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description 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.

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.

clone(*self*)

Constructs a new component with the same parameters and random state.

Returns A new instance of this component with identical parameters and random state.

default_parameters(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

Returns Default parameters for this component.

Return type dict

describe(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

Parameters

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description 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.

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

hyperparameter_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.

clone(*self*)

Constructs a new component with the same parameters and random state.

Returns A new instance of this component with identical parameters and random state.

default_parameters(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

Returns Default parameters for this component.

Return type dict

describe(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

Parameters

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description 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.

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

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.

clone(*self*)

Constructs a new component with the same parameters and random state.

Returns A new instance of this component with identical parameters and random state.

default_parameters(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

Returns Default parameters for this component.

Return type dict

describe(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

Parameters

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description 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.

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

hyperparameter_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.

clone(*self*)

Constructs a new component with the same parameters and random state.

Returns A new instance of this component with identical parameters and random state.

default_parameters(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

Returns Default parameters for this component.

Return type dict

describe(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

Parameters

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description 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.

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.

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

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.

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=1.0*, *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.

clone(*self*)

Constructs a new component with the same parameters and random state.

Returns A new instance of this component with identical parameters and random state.

default_parameters(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

Returns Default parameters for this component.

Return type dict

describe(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

Parameters

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description 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

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.
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

hyperparameter_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.

clone(*self*)

Constructs a new component with the same parameters and random state.

Returns A new instance of this component with identical parameters and random state.

default_parameters(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

Returns Default parameters for this component.

Return type dict

describe(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

Parameters

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description 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.

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

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.

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

class evalml.pipelines.components.transformers.encoders.**TargetEncoder**(*cols=None*, *smoothing=1.0*, *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.

clone(*self*)

Constructs a new component with the same parameters and random state.

Returns A new instance of this component with identical parameters and random state.

default_parameters(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

Returns Default parameters for this component.

Return type dict

describe(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

Parameters

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description 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

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_s
***kwa*

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.

clone(*self*)

Constructs a new component with the same parameters and random state.

Returns A new instance of this component with identical parameters and random state.

default_parameters(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

Returns Default parameters for this component.

Return type dict

describe(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

Parameters

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description 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

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 0.5. Defaults to None.
- **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.
- **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 -np.inf.
- **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.

clone(*self*)

Constructs a new component with the same parameters and random state.

Returns A new instance of this component with identical parameters and random state.

default_parameters(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

Returns Default parameters for this component.

Return type dict

describe(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

Parameters

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description 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

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. Defaults to None.
- **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.

- **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 -np.inf.
- **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 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.

clone(*self*)

Constructs a new component with the same parameters and random state.

Returns A new instance of this component with identical parameters and random state.

default_parameters(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

Returns Default parameters for this component.

Return type dict

`describe`(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

Parameters

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description 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

Package Contents

Classes Summary

FeatureSelector	Selects top features based on importance weights.
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.

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.

clone(*self*)

Constructs a new component with the same parameters and random state.

Returns A new instance of this component with identical parameters and random state.

default_parameters(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

Returns Default parameters for this component.

Return type dict

describe(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

Parameters

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description 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

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 0.5. Defaults to None.
- **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.
- **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 -np.inf.
- **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.

clone(*self*)

Constructs a new component with the same parameters and random state.

Returns A new instance of this component with identical parameters and random state.

default_parameters(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

Returns Default parameters for this component.

Return type dict

describe(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

Parameters

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description 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

class evalml.pipelines.components.transformers.feature_selection.**RFRegressorSelectFromModel**(*number_featur n_estimators= max_depth=No 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. Defaults to None.
- **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.
- **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 -np.inf.
- **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 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.

clone(*self*)

Constructs a new component with the same parameters and random state.

Returns A new instance of this component with identical parameters and random state.

default_parameters(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

Returns Default parameters for this component.

Return type dict

describe(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

Parameters

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description 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

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*
categorical_fill_value=None,
numeric_impute_strategy='mean',
numeric_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".
- **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.
- **random_seed** (*int*) – Seed for the random number generator. Defaults to 0.

Attributes

hyperparameter_ranges	{ "categorical_impute_strategy": ["most_frequent"], "numeric_impute_strategy": ["mean", "median", "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. 'None' values are converted to np.nan before imputation and are treated as the same.

clone(*self*)

Constructs a new component with the same parameters and random state.

Returns A new instance of this component with identical parameters and random state.

default_parameters(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

Returns Default parameters for this component.

Return type dict

describe(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

Parameters

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description 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. '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*) – Ignored.

Returns Transformed X

Return type pd.DataFrame

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_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

hyperparameter_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.

`clone`(*self*)

Constructs a new component with the same parameters and random state.

Returns A new instance of this component with identical parameters and random state.

`default_parameters`(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

Returns Default parameters for this component.

Return type dict

`describe`(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

Parameters

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description 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

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,
ran-
dom_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.

clone(*self*)

Constructs a new component with the same parameters and random state.

Returns A new instance of this component with identical parameters and random state.

default_parameters(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

Returns Default parameters for this component.

Return type dict

describe(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

Parameters

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description 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

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

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.

clone(*self*)

Constructs a new component with the same parameters and random state.

Returns A new instance of this component with identical parameters and random state.

default_parameters(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

Returns Default parameters for this component.

Return type dict

describe(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

Parameters

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description 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)

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']
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.

Package Contents

Classes Summary

Imputer	Imputes missing data according to a specified imputation strategy.
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.

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*, *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”.

- **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.
- **random_seed** (*int*) – Seed for the random number generator. Defaults to 0.

Attributes

hyperparameter_ranges	{ "categorical_impute_strategy": ["most_frequent"], "numeric_impute_strategy": ["mean", "median", "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. 'None' values are converted to np.nan before imputation and are treated as the same.

clone(*self*)

Constructs a new component with the same parameters and random state.

Returns A new instance of this component with identical parameters and random state.

default_parameters(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

Returns Default parameters for this component.

Return type dict

describe(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

Parameters

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description 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. '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*) – Ignored.

Returns Transformed X

Return type pd.DataFrame

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

hyperparameter_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.

clone(*self*)

Constructs a new component with the same parameters and random state.

Returns A new instance of this component with identical parameters and random state.

default_parameters(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

Returns Default parameters for this component.

Return type dict

describe(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

Parameters

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description 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

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

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.

clone(*self*)

Constructs a new component with the same parameters and random state.

Returns A new instance of this component with identical parameters and random state.

default_parameters(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

Returns Default parameters for this component.

Return type dict

describe(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

Parameters

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description 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

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

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

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.

clone(*self*)

Constructs a new component with the same parameters and random state.

Returns A new instance of this component with identical parameters and random state.

`default_parameters`(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

Returns Default parameters for this component.

Return type dict

`describe`(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

Parameters

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description 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)

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
***kv*

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

hyperparameter_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.

clone(*self*)

Constructs a new component with the same parameters and random state.

Returns A new instance of this component with identical parameters and random state.

default_parameters(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

Returns Default parameters for this component.

Return type dict

`describe`(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

Parameters

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description 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

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

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.

`clone`(*self*)

Constructs a new component with the same parameters and random state.

Returns A new instance of this component with identical parameters and random state.

`default_parameters`(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

Returns Default parameters for this component.

Return type dict

`describe`(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

Parameters

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description 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)

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.

clone(*self*)

Constructs a new component with the same parameters and random state.

Returns A new instance of this component with identical parameters and random state.

default_parameters(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

Returns Default parameters for this component.

Return type dict

describe(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

Parameters

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description 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

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.

clone(*self*)

Constructs a new component with the same parameters and random state.

Returns A new instance of this component with identical parameters and random state.

default_parameters(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

Returns Default parameters for this component.

Return type dict

describe(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

Parameters

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description 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)

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.

Attributes

hyperparameter_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.
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.

clone(*self*)

Constructs a new component with the same parameters and random state.

Returns A new instance of this component with identical parameters and random state.

default_parameters(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

Returns Default parameters for this component.

Return type dict

describe(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

Parameters

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description 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

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

hyperparameter_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.

clone(*self*)

Constructs a new component with the same parameters and random state.

Returns A new instance of this component with identical parameters and random state.

default_parameters(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

Returns Default parameters for this component.

Return type dict

describe(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

Parameters

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description 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

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

hyperparameter_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.

clone(*self*)

Constructs a new component with the same parameters and random state.

Returns A new instance of this component with identical parameters and random state.

default_parameters(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

Returns Default parameters for this component.

Return type dict

describe(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

Parameters

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description 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

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

hyperparameter_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.

clone(*self*)

Constructs a new component with the same parameters and random state.

Returns A new instance of this component with identical parameters and random state.

default_parameters(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

Returns Default parameters for this component.

Return type dict

describe(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

Parameters

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description 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

polynomial_detrender

Component that removes trends from time series by fitting a polynomial to the data.

Module Contents

Classes Summary

PolynomialDetrender	Removes trends from time series by fitting a polynomial to the data.

Contents

class evalml.pipelines.components.transformers.preprocessing.polynomial_detrender.**PolynomialDetrender**(*d*
r
d
***

Removes trends from time series by fitting a polynomial to the data.

Parameters

- **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.
- **random_seed** (*int*) – Seed for the random number generator. Defaults to 0.

Attributes

hyperparameter_ranges	{ "degree": Integer(1, 3)}
modifies_features	False
modifies_target	True
name	Polynomial Detrender
training_only	False

Methods

clone	Constructs a new 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 PolynomialDetrender.
fit_transform	Removes fitted trend from target variable.
inverse_transform	Adds back fitted trend to target variable.
load	Loads component at file path.
needs_fitting	Returns boolean determining 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	Removes fitted trend from target variable.

clone(*self*)

Constructs a new component with the same parameters and random state.

Returns A new instance of this component with identical parameters and random state.

default_parameters(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

Returns Default parameters for this component.

Return type dict

describe(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

Parameters

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description 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 PolynomialDetrender.

Parameters

- **X** (*pd.DataFrame, optional*) – Ignored.
- **y** (*pd.Series*) – Target variable to detrend.

Returns self

Raises **ValueError** – If y is None.

fit_transform(*self*, *X*, *y=None*)

Removes fitted trend from target variable.

Parameters

- **X** (*pd.DataFrame, optional*) – Ignored.
- **y** (*pd.Series*) – Target variable to detrend.

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

inverse_transform(*self*, *y*)

Adds back fitted trend to target variable.

Parameters **y** (*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 added back.

Return type tuple of pd.DataFrame, pd.Series

Raises **ValueError** – If 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*)

Removes fitted trend from target variable.

Parameters

- **X** (*pd.DataFrame, optional*) – Ignored.
- **y** (*pd.Series*) – Target variable to detrend.

Returns

The input features are returned without modification. The target variable y is detrended

Return type tuple of pd.DataFrame, pd.Series

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

hyperparameter_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.

clone(*self*)

Constructs a new component with the same parameters and random state.

Returns A new instance of this component with identical parameters and random state.

default_parameters(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

Returns Default parameters for this component.

Return type dict

describe(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

Parameters

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description 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

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_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.

clone(*self*)

Constructs a new component with the same parameters and random state.

Returns A new instance of this component with identical parameters and random state.

`default_parameters`(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

Returns Default parameters for this component.

Return type dict

`describe`(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

Parameters

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description 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.

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

hyperparameter_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.

clone(*self*)

Constructs a new component with the same parameters and random state.

Returns A new instance of this component with identical parameters and random state.

default_parameters(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

Returns Default parameters for this component.

Return type dict

describe(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

Parameters

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description 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

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.**EmailFeaturi**

Transformer that can automatically extract features from emails.

Parameters **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	Email Featurizer
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.

clone(*self*)

Constructs a new component with the same parameters and random state.

Returns A new instance of this component with identical parameters and random state.

default_parameters(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

Returns Default parameters for this component.

Return type dict

describe(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

Parameters

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description 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.

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

hyperparameter_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.

clone(*self*)

Constructs a new component with the same parameters and random state.

Returns A new instance of this component with identical parameters and random state.

default_parameters(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

Returns Default parameters for this component.

Return type dict

describe(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

Parameters

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description 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.

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.
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.
PolynomialDetrender	Removes trends from time series by fitting a polynomial to the data.
ReplaceNullableTypes	Transformer to replace features with the new nullable dtypes with a dtype that is compatible in EvalML.
TextTransformer	Base class for all transformers working with text features.
TimeSeriesFeaturizer	Transformer that delays input features and target variable for time series problems.
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

hyperparameter_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.

clone(*self*)

Constructs a new component with the same parameters and random state.

Returns A new instance of this component with identical parameters and random state.

default_parameters(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

Returns Default parameters for this component.

Return type dict

describe(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

Parameters

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description 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

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.

Attributes

hyperparameter_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.
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.

clone(*self*)

Constructs a new component with the same parameters and random state.

Returns A new instance of this component with identical parameters and random state.

default_parameters(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

Returns Default parameters for this component.

Return type dict

describe(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

Parameters

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description 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

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.

`clone`(*self*)

Constructs a new component with the same parameters and random state.

Returns A new instance of this component with identical parameters and random state.

`default_parameters`(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

Returns Default parameters for this component.

Return type dict

`describe`(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

Parameters

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description 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)

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

hyperparameter_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.

clone(*self*)

Constructs a new component with the same parameters and random state.

Returns A new instance of this component with identical parameters and random state.

default_parameters(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

Returns Default parameters for this component.

Return type dict

describe(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

Parameters

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description 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

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.

clone(*self*)

Constructs a new component with the same parameters and random state.

Returns A new instance of this component with identical parameters and random state.

default_parameters(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

Returns Default parameters for this component.

Return type dict

describe(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

Parameters

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description 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)

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

hyperparameter_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.

clone(*self*)

Constructs a new component with the same parameters and random state.

Returns A new instance of this component with identical parameters and random state.

default_parameters(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

Returns Default parameters for this component.

Return type dict

describe(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

Parameters

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description 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.

class evalml.pipelines.components.transformers.preprocessing.**LogTransformer**(*random_seed=0*)

Applies a log transformation to the target data.

Attributes

hyperparameter_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.

clone(*self*)

Constructs a new component with the same parameters and random state.

Returns A new instance of this component with identical parameters and random state.

default_parameters(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

Returns Default parameters for this component.

Return type dict

describe(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

Parameters

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description 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

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

hyperparameter_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.

clone(*self*)

Constructs a new component with the same parameters and random state.

Returns A new instance of this component with identical parameters and random state.

default_parameters(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

Returns Default parameters for this component.

Return type dict

describe(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

Parameters

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description 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

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

hyperparameter_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.

clone(*self*)

Constructs a new component with the same parameters and random state.

Returns A new instance of this component with identical parameters and random state.

default_parameters(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

Returns Default parameters for this component.

Return type dict

describe(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

Parameters

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description 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

class evalml.pipelines.components.transformers.preprocessing.**PolynomialDetrender**(*degree=1*, *random_seed=0*, ***kwargs*)

Removes trends from time series by fitting a polynomial to the data.

Parameters

- **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.
- **random_seed** (*int*) – Seed for the random number generator. Defaults to 0.

Attributes

hyperparameter_ranges	{ "degree": Integer(1, 3)}
modifies_features	False
modifies_target	True
name	Polynomial Detrender
training_only	False

Methods

clone	Constructs a new 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 PolynomialDetrender.
fit_transform	Removes fitted trend from target variable.
inverse_transform	Adds back fitted trend to target variable.
load	Loads component at file path.
needs_fitting	Returns boolean determining 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	Removes fitted trend from target variable.

clone(*self*)

Constructs a new component with the same parameters and random state.

Returns A new instance of this component with identical parameters and random state.

default_parameters(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

Returns Default parameters for this component.

Return type dict

describe(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

Parameters

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description 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 PolynomialDetrender.

Parameters

- **X** (*pd.DataFrame, optional*) – Ignored.
- **y** (*pd.Series*) – Target variable to detrend.

Returns self

Raises **ValueError** – If y is None.

fit_transform(*self*, *X*, *y=None*)

Removes fitted trend from target variable.

Parameters

- **X** (*pd.DataFrame, optional*) – Ignored.
- **y** (*pd.Series*) – Target variable to detrend.

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

inverse_transform(*self*, *y*)

Adds back fitted trend to target variable.

Parameters **y** (*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 added back.

Return type tuple of pd.DataFrame, pd.Series

Raises **ValueError** – If 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*)

Removes fitted trend from target variable.

Parameters

- **X** (*pd.DataFrame, optional*) – Ignored.
- **y** (*pd.Series*) – Target variable to detrend.

Returns

The input features are returned without modification. The target variable y is detrended

Return type tuple of pd.DataFrame, pd.Series

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

hyperparameter_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.

clone(*self*)

Constructs a new component with the same parameters and random state.

Returns A new instance of this component with identical parameters and random state.

default_parameters(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

Returns Default parameters for this component.

Return type dict

describe(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

Parameters

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description 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

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.

clone(*self*)

Constructs a new component with the same parameters and random state.

Returns A new instance of this component with identical parameters and random state.

default_parameters(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

Returns Default parameters for this component.

Return type dict

describe(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

Parameters

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description 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.

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

hyperparameter_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.

clone(*self*)

Constructs a new component with the same parameters and random state.

Returns A new instance of this component with identical parameters and random state.

default_parameters(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

Returns Default parameters for this component.

Return type dict

describe(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

Parameters

- **print_name** (*bool, optional*) – whether to print name of component

- **return_dict** (*bool, optional*) – whether to return description 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

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

hyperparameter_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.

clone(*self*)

Constructs a new component with the same parameters and random state.

Returns A new instance of this component with identical parameters and random state.

default_parameters(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

Returns Default parameters for this component.

Return type dict

describe(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

Parameters

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description 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.

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.

clone(*self*)

Constructs a new component with the same parameters and random state.

Returns A new instance of this component with identical parameters and random state.

default_parameters(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

Returns Default parameters for this component.

Return type dict

describe(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

Parameters

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description 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=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

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

hyperparameter_ranges	{}
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 sampling the data.

clone(*self*)

Constructs a new component with the same parameters and random state.

Returns A new instance of this component with identical parameters and random state.

default_parameters(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

Returns Default parameters for this component.

Return type dict

describe(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

Parameters

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description 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 sampling the data.

Parameters

- **X** (*pd.DataFrame*) – Training features.
- **y** (*pd.Series*) – Target.

Returns Transformed features and target.

Return type pd.DataFrame, pd.Series

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

hyperparameter_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.

clone(*self*)

Constructs a new component with the same parameters and random state.

Returns A new instance of this component with identical parameters and random state.

default_parameters(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

Returns Default parameters for this component.

Return type dict

describe(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

Parameters

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description 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

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

hyperparameter_ranges	{}
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 sampling the data.

clone(*self*)

Constructs a new component with the same parameters and random state.

Returns A new instance of this component with identical parameters and random state.

default_parameters(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

Returns Default parameters for this component.

Return type dict

describe(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

Parameters

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description 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 sampling the data.

Parameters

- **X** (*pd.DataFrame*) – Training features.
- **y** (*pd.Series*) – Target.

Returns Transformed features and target.

Return type pd.DataFrame, pd.Series

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

hyperparameter_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.

clone(*self*)

Constructs a new component with the same parameters and random state.

Returns A new instance of this component with identical parameters and random state.

default_parameters(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

Returns Default parameters for this component.

Return type dict

describe(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

Parameters

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description 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

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

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 component to 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.

clone(*self*)

Constructs a new component with the same parameters and random state.

Returns A new instance of this component with identical parameters and random state.

default_parameters(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

Returns Default parameters for this component.

Return type dict

describe(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

Parameters

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description 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 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

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 component to 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.

clone(*self*)

Constructs a new component with the same parameters and random state.

Returns A new instance of this component with identical parameters and random state.

default_parameters(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

Returns Default parameters for this component.

Return type dict

describe(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

Parameters

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description 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 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

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.

clone(*self*)

Constructs a new component with the same parameters and random state.

Returns A new instance of this component with identical parameters and random state.

default_parameters(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

Returns Default parameters for this component.

Return type dict

describe(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

Parameters

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description 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

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

hyperparameter_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.

clone(*self*)

Constructs a new component with the same parameters and random state.

Returns A new instance of this component with identical parameters and random state.

default_parameters(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

Returns Default parameters for this component.

Return type dict

describe(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

Parameters

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description 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

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

hyperparameter_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.

clone(*self*)

Constructs a new component with the same parameters and random state.

Returns A new instance of this component with identical parameters and random state.

default_parameters(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

Returns Default parameters for this component.

Return type dict

`describe`(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

Parameters

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description 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

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

hyperparameter_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.

clone(*self*)

Constructs a new component with the same parameters and random state.

Returns A new instance of this component with identical parameters and random state.

default_parameters(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

Returns Default parameters for this component.

Return type dict

describe(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

Parameters

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description 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

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.

clone(*self*)

Constructs a new component with the same parameters and random state.

Returns A new instance of this component with identical parameters and random state.

default_parameters(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

Returns Default parameters for this component.

Return type dict

describe(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

Parameters

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description 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.

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.
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.

continues on next page

Table 5 – continued from previous page

PolynomialDetrender	Removes trends from time series by fitting a polynomial to the data.
ReplaceNullableTypes	Transformer to replace features with the new nullable dtypes with a dtype that is compatible in EvalML.
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.
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.
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.
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

hyperparameter_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.

clone(*self*)

Constructs a new component with the same parameters and random state.

Returns A new instance of this component with identical parameters and random state.

default_parameters(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

Returns Default parameters for this component.

Return type dict

describe(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

Parameters

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description 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

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.

Attributes

hyperparameter_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.
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.

clone(*self*)

Constructs a new component with the same parameters and random state.

Returns A new instance of this component with identical parameters and random state.

default_parameters(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

Returns Default parameters for this component.

Return type dict

describe(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

Parameters

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description 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

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

hyperparameter_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.

clone(*self*)

Constructs a new component with the same parameters and random state.

Returns A new instance of this component with identical parameters and random state.

default_parameters(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

Returns Default parameters for this component.

Return type dict

describe(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

Parameters

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description 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

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

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.

clone(*self*)

Constructs a new component with the same parameters and random state.

Returns A new instance of this component with identical parameters and random state.

default_parameters(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

Returns Default parameters for this component.

Return type dict

describe(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

Parameters

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description 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)

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

hyperparameter_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.

clone(*self*)

Constructs a new component with the same parameters and random state.

Returns A new instance of this component with identical parameters and random state.

default_parameters(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

Returns Default parameters for this component.

Return type dict

describe(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

Parameters

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description 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

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

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.

clone(*self*)

Constructs a new component with the same parameters and random state.

Returns A new instance of this component with identical parameters and random state.

default_parameters(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

Returns Default parameters for this component.

Return type dict

describe(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

Parameters

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description 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)

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

hyperparameter_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.

clone(*self*)

Constructs a new component with the same parameters and random state.

Returns A new instance of this component with identical parameters and random state.

default_parameters(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

Returns Default parameters for this component.

Return type dict

describe(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

Parameters

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description 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.

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.

clone(*self*)

Constructs a new component with the same parameters and random state.

Returns A new instance of this component with identical parameters and random state.

default_parameters(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

Returns Default parameters for this component.

Return type dict

describe(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

Parameters

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description 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

class evalml.pipelines.components.transformers.**Imputer**(*categorical_impute_strategy='most_frequent'*, *categorical_fill_value=None*, *numeric_impute_strategy='mean'*, *numeric_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”.
- **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.
- **random_seed** (*int*) – Seed for the random number generator. Defaults to 0.

Attributes

hyperparameter_ranges	{ "categorical_impute_strategy": ["most_frequent"], "numeric_impute_strategy": ["mean", "median", "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. 'None' values are converted to np.nan before imputation and are treated as the same.

clone(*self*)

Constructs a new component with the same parameters and random state.

Returns A new instance of this component with identical parameters and random state.

default_parameters(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

Returns Default parameters for this component.

Return type dict

describe(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

Parameters

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description 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. '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*) – Ignored.

Returns Transformed X

Return type pd.DataFrame

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

hyperparameter_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.

clone(*self*)

Constructs a new component with the same parameters and random state.

Returns A new instance of this component with identical parameters and random state.

default_parameters(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

Returns Default parameters for this component.

Return type dict

describe(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

Parameters

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description 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.

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

hyperparameter_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.

clone(*self*)

Constructs a new component with the same parameters and random state.

Returns A new instance of this component with identical parameters and random state.

default_parameters(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

Returns Default parameters for this component.

Return type dict

describe(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

Parameters

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description 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.

class evalml.pipelines.components.transformers.**LogTransformer**(*random_seed=0*)

Applies a log transformation to the target data.

Attributes

hyperparameter_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.

clone(*self*)

Constructs a new component with the same parameters and random state.

Returns A new instance of this component with identical parameters and random state.

default_parameters(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

Returns Default parameters for this component.

Return type dict

describe(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

Parameters

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description 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

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

hyperparameter_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.

clone(*self*)

Constructs a new component with the same parameters and random state.

Returns A new instance of this component with identical parameters and random state.

default_parameters(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

Returns Default parameters for this component.

Return type dict

describe(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

Parameters

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description 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

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

hyperparameter_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.

clone(*self*)

Constructs a new component with the same parameters and random state.

Returns A new instance of this component with identical parameters and random state.

default_parameters(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

Returns Default parameters for this component.

Return type dict

describe(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

Parameters

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description 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

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

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.

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

`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

hyperparameter_ranges	{}
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 sampling the data.

clone(*self*)

Constructs a new component with the same parameters and random state.

Returns A new instance of this component with identical parameters and random state.

default_parameters(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

Returns Default parameters for this component.

Return type dict

describe(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

Parameters

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description 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 sampling the data.

Parameters

- **X** (*pd.DataFrame*) – Training features.
- **y** (*pd.Series*) – Target.

Returns Transformed features and target.

Return type pd.DataFrame, pd.Series

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

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.

clone(*self*)

Constructs a new component with the same parameters and random state.

Returns A new instance of this component with identical parameters and random state.

default_parameters(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

Returns Default parameters for this component.

Return type dict

describe(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

Parameters

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description 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.

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

hyperparameter_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.

clone(*self*)

Constructs a new component with the same parameters and random state.

Returns A new instance of this component with identical parameters and random state.

default_parameters(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

Returns Default parameters for this component.

Return type dict

describe(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

Parameters

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description 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

class evalml.pipelines.components.transformers.**PolynomialDetrender**(*degree=1*, *random_seed=0*, ***kwargs*)

Removes trends from time series by fitting a polynomial to the data.

Parameters

- **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.
- **random_seed** (*int*) – Seed for the random number generator. Defaults to 0.

Attributes

hyperparameter_ranges	{ "degree": Integer(1, 3)}
modifies_features	False
modifies_target	True
name	Polynomial Detrender
training_only	False

Methods

clone	Constructs a new 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 PolynomialDetrender.
fit_transform	Removes fitted trend from target variable.
inverse_transform	Adds back fitted trend to target variable.
load	Loads component at file path.
needs_fitting	Returns boolean determining 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	Removes fitted trend from target variable.

clone(*self*)

Constructs a new component with the same parameters and random state.

Returns A new instance of this component with identical parameters and random state.

default_parameters(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

Returns Default parameters for this component.

Return type dict

describe(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

Parameters

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description 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 PolynomialDetrender.

Parameters

- **X** (*pd.DataFrame, optional*) – Ignored.
- **y** (*pd.Series*) – Target variable to detrend.

Returns self

Raises **ValueError** – If y is None.

fit_transform(*self*, *X*, *y=None*)

Removes fitted trend from target variable.

Parameters

- **X** (*pd.DataFrame, optional*) – Ignored.
- **y** (*pd.Series*) – Target variable to detrend.

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

inverse_transform(*self*, *y*)

Adds back fitted trend to target variable.

Parameters **y** (*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 added back.

Return type tuple of pd.DataFrame, pd.Series

Raises **ValueError** – If 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*)

Removes fitted trend from target variable.

Parameters

- **X** (*pd.DataFrame, optional*) – Ignored.
- **y** (*pd.Series*) – Target variable to detrend.

Returns

The input features are returned without modification. The target variable y is detrended

Return type tuple of pd.DataFrame, pd.Series

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

hyperparameter_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.

clone(*self*)

Constructs a new component with the same parameters and random state.

Returns A new instance of this component with identical parameters and random state.

default_parameters(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

Returns Default parameters for this component.

Return type dict

describe(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

Parameters

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description 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

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 0.5. Defaults to None.
- **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.
- **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 -np.inf.
- **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.

clone(*self*)

Constructs a new component with the same parameters and random state.

Returns A new instance of this component with identical parameters and random state.

default_parameters(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

Returns Default parameters for this component.

Return type dict

describe(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

Parameters

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description 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

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. Defaults to None.
- **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.
- **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 -np.inf.
- **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 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.

clone(*self*)

Constructs a new component with the same parameters and random state.

Returns A new instance of this component with identical parameters and random state.

default_parameters(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

Returns Default parameters for this component.

Return type dict

describe(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

Parameters

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description 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

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

hyperparameter_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.

clone(*self*)

Constructs a new component with the same parameters and random state.

Returns A new instance of this component with identical parameters and random state.

default_parameters(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

Returns Default parameters for this component.

Return type dict

describe(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

Parameters

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description 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

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

hyperparameter_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.

clone(*self*)

Constructs a new component with the same parameters and random state.

Returns A new instance of this component with identical parameters and random state.

default_parameters(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

Returns Default parameters for this component.

Return type dict

describe(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

Parameters

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description 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

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

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.

clone(*self*)

Constructs a new component with the same parameters and random state.

Returns A new instance of this component with identical parameters and random state.

default_parameters(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

Returns Default parameters for this component.

Return type dict

describe(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

Parameters

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description 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

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

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

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 component to 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.

clone(*self*)

Constructs a new component with the same parameters and random state.

Returns A new instance of this component with identical parameters and random state.

default_parameters(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

Returns Default parameters for this component.

Return type dict

describe(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

Parameters

- **print_name** (*bool, optional*) – whether to print name of component

- **return_dict** (*bool, optional*) – whether to return description 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 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

class evalml.pipelines.components.transformers.**TargetEncoder**(*cols=None*, *smoothing=1.0*, *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.

clone(*self*)

Constructs a new component with the same parameters and random state.

Returns A new instance of this component with identical parameters and random state.

default_parameters(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

Returns Default parameters for this component.

Return type dict

describe(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

Parameters

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description 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

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.

clone(*self*)

Constructs a new component with the same parameters and random state.

Returns A new instance of this component with identical parameters and random state.

default_parameters(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

Returns Default parameters for this component.

Return type dict

describe(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

Parameters

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description 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)

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

hyperparameter_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.

clone(*self*)

Constructs a new component with the same parameters and random state.

Returns A new instance of this component with identical parameters and random state.

default_parameters(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

Returns Default parameters for this component.

Return type dict

describe(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

Parameters

- **print_name** (*bool, optional*) – whether to print name of component

- **return_dict** (*bool, optional*) – whether to return description 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

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.

clone(*self*)

Constructs a new component with the same parameters and random state.

Returns A new instance of this component with identical parameters and random state.

default_parameters(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

Returns Default parameters for this component.

Return type dict

describe(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

Parameters

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description 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.

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

hyperparameter_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.

clone(*self*)

Constructs a new component with the same parameters and random state.

Returns A new instance of this component with identical parameters and random state.

default_parameters(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

Returns Default parameters for this component.

Return type dict

`describe`(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

Parameters

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description 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

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

hyperparameter_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.

clone(*self*)

Constructs a new component with the same parameters and random state.

Returns A new instance of this component with identical parameters and random state.

default_parameters(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

Returns Default parameters for this component.

Return type dict

describe(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

Parameters

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description 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.

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.

clone(*self*)

Constructs a new component with the same parameters and random state.

Returns A new instance of this component with identical parameters and random state.

default_parameters(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

Returns Default parameters for this component.

Return type dict

describe(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

Parameters

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description 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.

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’]
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.
handle_component_class	Standardizes input from a string name to a ComponentBase subclass if necessary.
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': 'mse', '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.**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.**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)` wrt. *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) wrt. *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.

continues on next page

Table 6 – continued from previous page

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.
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.
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.
PolynomialDetrender	Removes trends from time series by fitting a polynomial to the data.

continues on next page

Table 6 – continued from previous page

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.
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.
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.
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.
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.
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=None*, *trend=None*, *start_p=2*, *d=0*, *start_q=2*, *max_p=5*, *max_d=2*, *max_q=5*, *seasonal=True*, *n_jobs=- 1*, *random_seed=0*, *maxiter=10*, ***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.
- **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

hyperparameter_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],}
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.
load	Loads component at file path.
needs_fitting	Returns boolean determining 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.

clone(*self*)

Constructs a new component with the same parameters and random state.

Returns A new instance of this component with identical parameters and random state.

default_parameters(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

Returns Default parameters for this component.

Return type dict

describe(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

Parameters

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description 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 with a length of 1 as feature_importance is not defined for ARIMA regressor.

fit(*self*, *X*, *y=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 X was passed to *fit* but not passed in *predict*.

static load(*file_path*)

Loads component at file path.

Parameters **file_path** (*str*) – Location to load file.

Returns ComponentBase object

needs_fitting(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need 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*, *y=None*)

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*)

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.

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

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.
load	Loads component at file path.
needs_fitting	Returns boolean determining 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.

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.

static load(*file_path*)

Loads component at file path.

Parameters **file_path** (*str*) – Location to load file.

Returns ComponentBase object

needs_fitting(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need 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.

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

hyperparameter_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.
load	Loads component at file path.
needs_fitting	Returns boolean determining 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.

clone(*self*)

Constructs a new component with the same parameters and random state.

Returns A new instance of this component with identical parameters and random state.

default_parameters(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

Returns Default parameters for this component.

Return type dict

describe(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

Parameters

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description 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.

static load(*file_path*)

Loads component at file path.

Parameters **file_path** (*str*) – Location to load file.

Returns ComponentBase object

needs_fitting(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need 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*)

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.

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.
load	Loads component at file path.
needs_fitting	Returns boolean determining 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 probability estimates for labels.
save	Saves component at file path.

clone(*self*)

Constructs a new component with the same parameters and random state.

Returns A new instance of this component with identical parameters and random state.

default_parameters(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

Returns Default parameters for this component.

Return type dict

describe(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

Parameters

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description 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

static load(*file_path*)

Loads component at file path.

Parameters **file_path** (*str*) – Location to load file.

Returns ComponentBase object

needs_fitting(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need 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.DataFrame

predict_proba(*self*, *X*)

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.

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

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 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.
load	Loads component at file path.
needs_fitting	Returns boolean determining 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.

clone(*self*)

Constructs a new component with the same parameters and random state.

Returns A new instance of this component with identical parameters and random state.

default_parameters(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

Returns Default parameters for this component.

Return type dict

describe(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

Parameters

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description 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

static load(*file_path*)

Loads component at file path.

Parameters **file_path** (*str*) – Location to load file.

Returns ComponentBase object

needs_fitting(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need 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 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*)

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.

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.

clone(*self*)

Constructs a new component with the same parameters and random state.

Returns A new instance of this component with identical parameters and random state.

default_parameters(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

Returns Default parameters for this component.

Return type dict

describe(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

Parameters

- **print_name** (*bool, optional*) – whether to print name of component

- **return_dict** (*bool, optional*) – whether to return description 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.

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’]
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

hyperparameter_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.

clone(*self*)

Constructs a new component with the same parameters and random state.

Returns A new instance of this component with identical parameters and random state.

default_parameters(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

Returns Default parameters for this component.

Return type dict

describe(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

Parameters

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description 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

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

hyperparameter_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.
load	Loads component at file path.
needs_fitting	Returns boolean determining 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.

`clone`(*self*)

Constructs a new component with the same parameters and random state.

Returns A new instance of this component with identical parameters and random state.

`default_parameters`(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

Returns Default parameters for this component.

Return type dict

`describe`(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

Parameters

- **print_name** (*bool, optional*) – whether to print name of component

- **return_dict** (*bool, optional*) – whether to return description 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.

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 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

static load(*file_path*)

Loads component at file path.

Parameters **file_path** (*str*) – Location to load file.

Returns ComponentBase object

needs_fitting(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need 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 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*)

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.

class evalml.pipelines.components.**DecisionTreeRegressor**(*criterion='mse'*, *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** (*{"mse", "friedman_mse", "mae", "poisson"}*) – The function to measure the quality of a split. Supported criteria are:
  - "mse" 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
  - "mae" 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": ["mse", "friedman_mse", "mae"], "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.
load	Loads component at file path.
needs_fitting	Returns boolean determining 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.

clone(*self*)

Constructs a new component with the same parameters and random state.

Returns A new instance of this component with identical parameters and random state.

default_parameters(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

Returns Default parameters for this component.

Return type dict

describe(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

Parameters

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description 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.

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 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

static load(*file_path*)

Loads component at file path.

Parameters **file_path** (*str*) – Location to load file.

Returns ComponentBase object

needs_fitting(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need 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 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*)

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.

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.

Attributes

hyperparameter_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.
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.

clone(*self*)

Constructs a new component with the same parameters and random state.

Returns A new instance of this component with identical parameters and random state.

default_parameters(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

Returns Default parameters for this component.

Return type dict

describe(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

Parameters

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description 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

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

hyperparameter_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.

clone(*self*)

Constructs a new component with the same parameters and random state.

Returns A new instance of this component with identical parameters and random state.

default_parameters(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

Returns Default parameters for this component.

Return type dict

describe(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

Parameters

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description 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

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

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.

`clone`(*self*)

Constructs a new component with the same parameters and random state.

Returns A new instance of this component with identical parameters and random state.

`default_parameters`(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

Returns Default parameters for this component.

Return type dict

`describe`(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

Parameters

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description 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)

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

hyperparameter_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.

clone(*self*)

Constructs a new component with the same parameters and random state.

Returns A new instance of this component with identical parameters and random state.

default_parameters(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

Returns Default parameters for this component.

Return type dict

describe(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

Parameters

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description 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

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

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.

clone(*self*)

Constructs a new component with the same parameters and random state.

Returns A new instance of this component with identical parameters and random state.

`default_parameters`(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

Returns Default parameters for this component.

Return type dict

`describe`(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

Parameters

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description 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)

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

hyperparameter_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.
load	Loads component at file path.
needs_fitting	Returns boolean determining 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.

clone(*self*)

Constructs a new component with the same parameters and random state.

Returns A new instance of this component with identical parameters and random state.

default_parameters(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

Returns Default parameters for this component.

Return type dict

describe(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

Parameters

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description 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

static load(*file_path*)

Loads component at file path.

Parameters **file_path** (*str*) – Location to load file.

Returns ComponentBase object

needs_fitting(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need 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 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*)

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.

class evalml.pipelines.components.**ElasticNetRegressor**(*alpha=0.0001*, *l1_ratio=0.15*, *max_iter=1000*, *normalize=False*, *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.
- **normalize** (*boolean*) – If True, the regressors will be normalized before regression by subtracting the mean and dividing by the l2-norm. Defaults to False.
- **random_seed** (*int*) – Seed for the random number generator. Defaults to 0.

Attributes

hyperparameter_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.
load	Loads component at file path.
needs_fitting	Returns boolean determining 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.

clone(*self*)

Constructs a new component with the same parameters and random state.

Returns A new instance of this component with identical parameters and random state.

default_parameters(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

Returns Default parameters for this component.

Return type dict

describe(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

Parameters

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description 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*, *y=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

static load(*file_path*)

Loads component at file path.

Parameters **file_path** (*str*) – Location to load file.

Returns ComponentBase object

needs_fitting(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need 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 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*)

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.

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

hyperparameter_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.

clone(*self*)

Constructs a new component with the same parameters and random state.

Returns A new instance of this component with identical parameters and random state.

default_parameters(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

Returns Default parameters for this component.

Return type dict

describe(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

Parameters

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description 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.

class evalml.pipelines.components.**Estimator**(*parameters=None*, *component_obj=None*, *random_seed=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.
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.

clone(*self*)

Constructs a new component with the same parameters and random state.

Returns A new instance of this component with identical parameters and random state.

default_parameters(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

Returns Default parameters for this component.

Return type dict

describe(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

Parameters

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description 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.

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 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

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*)

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*)

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.

class evalml.pipelines.components.**ExponentialSmoothingRegressor**(*trend=None*, *damped_trend=False*, *seasonal=None*, *sp=2*, *n_jobs=-1*, *random_seed=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

hyperparameter_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.
load	Loads component at file path.
needs_fitting	Returns boolean determining 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.

clone(*self*)

Constructs a new component with the same parameters and random state.

Returns A new instance of this component with identical parameters and random state.

default_parameters(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

Returns Default parameters for this component.

Return type dict

describe(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

Parameters

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description 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 with a length of 1 as feature_importance is not defined for Exponential Smoothing regressor.

fit(*self*, *X*, *y=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.

static load(*file_path*)

Loads component at file path.

Parameters **file_path** (*str*) – Location to load file.

Returns ComponentBase object

needs_fitting(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need 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*, *y=None*)

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*)

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.

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

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.
load	Loads component at file path.
needs_fitting	Returns boolean determining 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.

clone(*self*)

Constructs a new component with the same parameters and random state.

Returns A new instance of this component with identical parameters and random state.

default_parameters(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

Returns Default parameters for this component.

Return type dict

describe(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

Parameters

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description 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.

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 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

static load(*file_path*)

Loads component at file path.

Parameters **file_path** (*str*) – Location to load file.

Returns ComponentBase object

needs_fitting(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need 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 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*)

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.

class evalml.pipelines.components.**ExtraTreesRegressor**(*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 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

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 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.
load	Loads component at file path.
needs_fitting	Returns boolean determining 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.

clone(*self*)

Constructs a new component with the same parameters and random state.

Returns A new instance of this component with identical parameters and random state.

default_parameters(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

Returns Default parameters for this component.

Return type dict

describe(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

Parameters

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description 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.

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 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

static load(*file_path*)

Loads component at file path.

Parameters **file_path** (*str*) – Location to load file.

Returns ComponentBase object

needs_fitting(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need 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 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*)

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.

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.

clone(*self*)

Constructs a new component with the same parameters and random state.

Returns A new instance of this component with identical parameters and random state.

default_parameters(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

Returns Default parameters for this component.

Return type dict

describe(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

Parameters

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description 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

class evalml.pipelines.components.**Imputer**(*categorical_impute_strategy='most_frequent'*, *categorical_fill_value=None*, *numeric_impute_strategy='mean'*, *numeric_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".
- **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.
- **random_seed** (*int*) – Seed for the random number generator. Defaults to 0.

Attributes

hyperparameter_ranges	{ "categorical_impute_strategy": ["most_frequent"], "numeric_impute_strategy": ["mean", "median", "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. 'None' values are converted to np.nan before imputation and are treated as the same.

clone(*self*)

Constructs a new component with the same parameters and random state.

Returns A new instance of this component with identical parameters and random state.

default_parameters(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

Returns Default parameters for this component.

Return type dict

describe(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

Parameters

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description 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. '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*) – Ignored.

Returns Transformed X

Return type pd.DataFrame

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

hyperparameter_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.
load	Loads component at file path.
needs_fitting	Returns boolean determining 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.

clone(*self*)

Constructs a new component with the same parameters and random state.

Returns A new instance of this component with identical parameters and random state.

default_parameters(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

Returns Default parameters for this component.

Return type dict

describe(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

Parameters

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description 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*, *y=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

static load(*file_path*)

Loads component at file path.

Parameters **file_path** (*str*) – Location to load file.

Returns ComponentBase object

needs_fitting(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need 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 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*)

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.

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

hyperparameter_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.

clone(*self*)

Constructs a new component with the same parameters and random state.

Returns A new instance of this component with identical parameters and random state.

default_parameters(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

Returns Default parameters for this component.

Return type dict

describe(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

Parameters

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description 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.

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

hyperparameter_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.
load	Loads component at file path.
needs_fitting	Returns boolean determining 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.

clone(*self*)

Constructs a new component with the same parameters and random state.

Returns A new instance of this component with identical parameters and random state.

default_parameters(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

Returns Default parameters for this component.

Return type dict

describe(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

Parameters

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description 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.

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

static load(*file_path*)

Loads component at file path.

Parameters **file_path** (*str*) – Location to load file.

Returns ComponentBase object

needs_fitting(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need 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.

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

hyperparameter_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.
load	Loads component at file path.
needs_fitting	Returns boolean determining 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.

clone(*self*)

Constructs a new component with the same parameters and random state.

Returns A new instance of this component with identical parameters and random state.

default_parameters(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

Returns Default parameters for this component.

Return type dict

describe(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

Parameters

- **print_name** (*bool, optional*) – whether to print name of component

- **return_dict** (*bool, optional*) – whether to return description 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.

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

static load(*file_path*)

Loads component at file path.

Parameters **file_path** (*str*) – Location to load file.

Returns ComponentBase object

needs_fitting(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need 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*)

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.

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

hyperparameter_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.

clone(*self*)

Constructs a new component with the same parameters and random state.

Returns A new instance of this component with identical parameters and random state.

default_parameters(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

Returns Default parameters for this component.

Return type dict

describe(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

Parameters

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description 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.

class evalml.pipelines.components.**LinearRegressor**(*fit_intercept=True*, *normalize=False*, *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.
- **normalize** (*boolean*) – If True, the regressors will be normalized before regression by subtracting the mean and dividing by the l2-norm. This parameter is ignored when fit_intercept is set to False. Defaults to False.
- **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], “normalize”: [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.
load	Loads component at file path.
needs_fitting	Returns boolean determining 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.

clone(*self*)

Constructs a new component with the same parameters and random state.

Returns A new instance of this component with identical parameters and random state.

default_parameters(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

Returns Default parameters for this component.

Return type dict

describe(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

Parameters

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description 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*, *y=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

static load(*file_path*)

Loads component at file path.

Parameters **file_path** (*str*) – Location to load file.

Returns ComponentBase object

needs_fitting(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need 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 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*)

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.

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

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.
load	Loads component at file path.
needs_fitting	Returns boolean determining 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.

clone(*self*)

Constructs a new component with the same parameters and random state.

Returns A new instance of this component with identical parameters and random state.

default_parameters(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

Returns Default parameters for this component.

Return type dict

describe(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

Parameters

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description 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*, *y=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

static load(*file_path*)

Loads component at file path.

Parameters **file_path** (*str*) – Location to load file.

Returns ComponentBase object

needs_fitting(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need 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 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*)

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.

class evalml.pipelines.components.**LogTransformer**(*random_seed=0*)

Applies a log transformation to the target data.

Attributes

hyperparameter_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.

clone(*self*)

Constructs a new component with the same parameters and random state.

Returns A new instance of this component with identical parameters and random state.

default_parameters(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

Returns Default parameters for this component.

Return type dict

describe(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

Parameters

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description 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

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

hyperparameter_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.

clone(*self*)

Constructs a new component with the same parameters and random state.

Returns A new instance of this component with identical parameters and random state.

default_parameters(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

Returns Default parameters for this component.

Return type dict

describe(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

Parameters

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description 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

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

hyperparameter_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.

clone(*self*)

Constructs a new component with the same parameters and random state.

Returns A new instance of this component with identical parameters and random state.

default_parameters(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

Returns Default parameters for this component.

Return type dict

describe(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

Parameters

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description 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

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

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.

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

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

hyperparameter_ranges	{}
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 sampling the data.

clone(*self*)

Constructs a new component with the same parameters and random state.

Returns A new instance of this component with identical parameters and random state.

default_parameters(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

Returns Default parameters for this component.

Return type dict

describe(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

Parameters

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description 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 sampling the data.

Parameters

- **X** (*pd.DataFrame*) – Training features.
- **y** (*pd.Series*) – Target.

Returns Transformed features and target.

Return type pd.DataFrame, pd.Series

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

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.

clone(*self*)

Constructs a new component with the same parameters and random state.

Returns A new instance of this component with identical parameters and random state.

default_parameters(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

Returns Default parameters for this component.

Return type dict

describe(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

Parameters

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description 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.

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

hyperparameter_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.

clone(*self*)

Constructs a new component with the same parameters and random state.

Returns A new instance of this component with identical parameters and random state.

default_parameters(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

Returns Default parameters for this component.

Return type dict

describe(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

Parameters

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description 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

class evalml.pipelines.components.**PolynomialDetrender**(*degree=1*, *random_seed=0*, ***kwargs*)

Removes trends from time series by fitting a polynomial to the data.

Parameters

- **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.
- **random_seed** (*int*) – Seed for the random number generator. Defaults to 0.

Attributes

hyperparameter_ranges	{ "degree": Integer(1, 3)}
modifies_features	False
modifies_target	True
name	Polynomial Detrender
training_only	False

Methods

clone	Constructs a new 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 PolynomialDetrender.
fit_transform	Removes fitted trend from target variable.
inverse_transform	Adds back fitted trend to target variable.
load	Loads component at file path.
needs_fitting	Returns boolean determining 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	Removes fitted trend from target variable.

clone(*self*)

Constructs a new component with the same parameters and random state.

Returns A new instance of this component with identical parameters and random state.

default_parameters(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

Returns Default parameters for this component.

Return type dict

describe(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

Parameters

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description 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 PolynomialDetrender.

Parameters

- **X** (*pd.DataFrame, optional*) – Ignored.
- **y** (*pd.Series*) – Target variable to detrend.

Returns self

Raises **ValueError** – If y is None.

fit_transform(*self*, *X*, *y=None*)

Removes fitted trend from target variable.

Parameters

- **X** (*pd.DataFrame, optional*) – Ignored.
- **y** (*pd.Series*) – Target variable to detrend.

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

inverse_transform(*self*, *y*)

Adds back fitted trend to target variable.

Parameters **y** (*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 added back.

Return type tuple of pd.DataFrame, pd.Series

Raises **ValueError** – If 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*)

Removes fitted trend from target variable.

Parameters

- **X** (*pd.DataFrame, optional*) – Ignored.
- **y** (*pd.Series*) – Target variable to detrend.

Returns

The input features are returned without modification. The target variable y is detrended

Return type tuple of pd.DataFrame, pd.Series

class evalml.pipelines.components.**ProphetRegressor**(*time_index=None*, *changepoint_prior_scale=0.05*, *seasonality_prior_scale=10*, *holidays_prior_scale=10*, *seasonality_mode='additive'*, *random_seed=0*, *stan_backend='CMDSTANPY'*, ***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/

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.
load	Loads component at file path.
needs_fitting	Returns boolean determining 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.

static build_prophet_df(*X*, *y=None*, *time_index='ds'*)

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*)

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*)

Returns array of 0's with len(1) as feature_importance is not defined for Prophet regressor.

fit(*self*, *X*, *y=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*)

Get parameters for the Prophet regressor.

static load(*file_path*)

Loads component at file path.

Parameters **file_path** (*str*) – Location to load file.

Returns ComponentBase object

needs_fitting(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need 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*, *y=None*)

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*)

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.

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

hyperparameter_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.
load	Loads component at file path.
needs_fitting	Returns boolean determining 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.

clone(*self*)

Constructs a new component with the same parameters and random state.

Returns A new instance of this component with identical parameters and random state.

default_parameters(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

Returns Default parameters for this component.

Return type dict

describe(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

Parameters

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description 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.

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 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

static load(*file_path*)

Loads component at file path.

Parameters **file_path** (*str*) – Location to load file.

Returns ComponentBase object

needs_fitting(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need 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 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*)

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.

class evalml.pipelines.components.**RandomForestRegressor**(*n_estimators=100*, *max_depth=6*, *n_jobs=-1*, *random_seed=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.
load	Loads component at file path.
needs_fitting	Returns boolean determining 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.

clone(*self*)

Constructs a new component with the same parameters and random state.

Returns A new instance of this component with identical parameters and random state.

default_parameters(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

Returns Default parameters for this component.

Return type dict

describe(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

Parameters

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description 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.

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 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

static load(*file_path*)

Loads component at file path.

Parameters **file_path** (*str*) – Location to load file.

Returns ComponentBase object

needs_fitting(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need 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 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*)

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.

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

hyperparameter_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.

clone(*self*)

Constructs a new component with the same parameters and random state.

Returns A new instance of this component with identical parameters and random state.

default_parameters(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

Returns Default parameters for this component.

Return type dict

describe(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

Parameters

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description 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

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 0.5. Defaults to None.
- **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.
- **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 -np.inf.
- **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.

clone(*self*)

Constructs a new component with the same parameters and random state.

Returns A new instance of this component with identical parameters and random state.

default_parameters(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

Returns Default parameters for this component.

Return type dict

describe(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

Parameters

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description 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

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. Defaults to None.
- **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.
- **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 -np.inf.
- **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 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.

clone(*self*)

Constructs a new component with the same parameters and random state.

Returns A new instance of this component with identical parameters and random state.

default_parameters(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

Returns Default parameters for this component.

Return type dict

describe(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

Parameters

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description 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

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

hyperparameter_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.

clone(*self*)

Constructs a new component with the same parameters and random state.

Returns A new instance of this component with identical parameters and random state.

default_parameters(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

Returns Default parameters for this component.

Return type dict

describe(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

Parameters

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description 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

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

hyperparameter_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.

clone(*self*)

Constructs a new component with the same parameters and random state.

Returns A new instance of this component with identical parameters and random state.

default_parameters(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

Returns Default parameters for this component.

Return type dict

describe(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

Parameters

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description 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

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

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.

clone(*self*)

Constructs a new component with the same parameters and random state.

Returns A new instance of this component with identical parameters and random state.

default_parameters(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

Returns Default parameters for this component.

Return type dict

describe(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

Parameters

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description 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

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

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"],
...     "Stacked Ensemble": [
...         StackedEnsembleClassifier(n_jobs=1, final_
↪estimator=DecisionTreeClassifier()),
...         "Decision Tree.x",
...         "Decision Tree B.x",
```

(continues on next page)

(continued from previous page)

```
...         "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

hyperparameter_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.
load	Loads component at file path.
needs_fitting	Returns boolean determining 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.

clone(*self*)

Constructs a new component with the same parameters 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*, *y=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

`static load`(*file_path*)

Loads component at file path.

Parameters **file_path** (*str*) – Location to load file.

Returns ComponentBase object

`needs_fitting`(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need 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 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*)

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.

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()),
...         "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

hyperparameter_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.
load	Loads component at file path.
needs_fitting	Returns boolean determining 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.

`clone`(*self*)

Constructs a new component with the same parameters 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*, *y=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

static load(*file_path*)

Loads component at file path.

Parameters **file_path** (*str*) – Location to load file.

Returns ComponentBase object

needs_fitting(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need 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 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*)

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.

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

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 component to 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.

clone(*self*)

Constructs a new component with the same parameters and random state.

Returns A new instance of this component with identical parameters and random state.

default_parameters(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

Returns Default parameters for this component.

Return type dict

describe(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

Parameters

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description 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 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

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

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.
load	Loads component at file path.
needs_fitting	Returns boolean determining 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.

clone(*self*)

Constructs a new component with the same parameters and random state.

Returns A new instance of this component with identical parameters and random state.

default_parameters(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

Returns Default parameters for this component.

Return type dict

describe(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

Parameters

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description 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, y=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

`static load(file_path)`

Loads component at file path.

Parameters **file_path** (*str*) – Location to load file.

Returns ComponentBase object

`needs_fitting(self)`

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need 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 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)`

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.

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

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.
load	Loads component at file path.
needs_fitting	Returns boolean determining 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.

clone(*self*)

Constructs a new component with the same parameters and random state.

Returns A new instance of this component with identical parameters and random state.

default_parameters(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

Returns Default parameters for this component.

Return type dict

describe(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

Parameters

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description 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*, *y=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

static load(*file_path*)

Loads component at file path.

Parameters **file_path** (*str*) – Location to load file.

Returns ComponentBase object

needs_fitting(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need 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 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*)

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.

class evalml.pipelines.components.**TargetEncoder**(*cols=None*, *smoothing=1.0*, *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.

clone(*self*)

Constructs a new component with the same parameters and random state.

Returns A new instance of this component with identical parameters and random state.

default_parameters(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

Returns Default parameters for this component.

Return type dict

describe(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

Parameters

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description 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

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

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.

clone(*self*)

Constructs a new component with the same parameters and random state.

Returns A new instance of this component with identical parameters and random state.

default_parameters(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

Returns Default parameters for this component.

Return type dict

describe(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

Parameters

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description 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)

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

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.
load	Loads component at file path.
needs_fitting	Returns boolean determining 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.

clone(*self*)

Constructs a new component with the same parameters and random state.

Returns A new instance of this component with identical parameters and random state.

default_parameters(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

Returns Default parameters for this component.

Return type dict

describe(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

Parameters

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description 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.

static load(*file_path*)

Loads component at file path.

Parameters **file_path** (*str*) – Location to load file.

Returns ComponentBase object

needs_fitting(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need 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.

`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

hyperparameter_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.

clone(*self*)

Constructs a new component with the same parameters and random state.

Returns A new instance of this component with identical parameters and random state.

default_parameters(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

Returns Default parameters for this component.

Return type dict

describe(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

Parameters

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description 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

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.

clone(*self*)

Constructs a new component with the same parameters and random state.

Returns A new instance of this component with identical parameters and random state.

default_parameters(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

Returns Default parameters for this component.

Return type dict

describe(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

Parameters

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description 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.

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

hyperparameter_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.

clone(*self*)

Constructs a new component with the same parameters and random state.

Returns A new instance of this component with identical parameters and random state.

default_parameters(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

Returns Default parameters for this component.

Return type dict

describe(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

Parameters

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description 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

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

hyperparameter_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.

clone(*self*)

Constructs a new component with the same parameters and random state.

Returns A new instance of this component with identical parameters and random state.

default_parameters(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

Returns Default parameters for this component.

Return type dict

describe(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

Parameters

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description 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.

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

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.
load	Loads component at file path.
needs_fitting	Returns boolean determining 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.

clone(*self*)

Constructs a new component with the same parameters and random state.

Returns A new instance of this component with identical parameters and random state.

default_parameters(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

Returns Default parameters for this component.

Return type dict

describe(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

Parameters

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description 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*, *y=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

static load(*file_path*)

Loads component at file path.

Parameters **file_path** (*str*) – Location to load file.

Returns ComponentBase object

needs_fitting(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need 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 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*)

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.

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

hyperparameter_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.
load	Loads component at file path.
needs_fitting	Returns boolean determining 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.

clone(*self*)

Constructs a new component with the same parameters and random state.

Returns A new instance of this component with identical parameters and random state.

default_parameters(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

Returns Default parameters for this component.

Return type dict

describe(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

Parameters

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description 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*, *y=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

static load(*file_path*)

Loads component at file path.

Parameters **file_path** (*str*) – Location to load file.

Returns ComponentBase object

needs_fitting(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need 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 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*)

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.

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

hyperparameter_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.
load	Loads component at file path.
needs_fitting	Returns boolean determining 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.

clone(*self*)

Constructs a new component with the same parameters and random state.

Returns A new instance of this component with identical parameters and random state.

default_parameters(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

Returns Default parameters for this component.

Return type dict

describe(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

Parameters

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description 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*, *y=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

static load(*file_path*)

Loads component at file path.

Parameters **file_path** (*str*) – Location to load file.

Returns ComponentBase object

needs_fitting(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need 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 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*)

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.

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

hyperparameter_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.
load	Loads component at file path.
needs_fitting	Returns boolean determining 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.

clone(*self*)

Constructs a new component with the same parameters and random state.

Returns A new instance of this component with identical parameters and random state.

default_parameters(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

Returns Default parameters for this component.

Return type dict

describe(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

Parameters

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description 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

static load(*file_path*)

Loads component at file path.

Parameters **file_path** (*str*) – Location to load file.

Returns ComponentBase object

needs_fitting(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need 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.

class evalml.pipelines.components.**XGBoostRegressor**(*eta=0.1*, *max_depth=6*, *min_child_weight=1*, *n_estimators=100*, *random_seed=0*, *n_jobs=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

hyperparameter_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.
load	Loads component at file path.
needs_fitting	Returns boolean determining 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.

clone(*self*)

Constructs a new component with the same parameters and random state.

Returns A new instance of this component with identical parameters and random state.

default_parameters(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

Returns Default parameters for this component.

Return type dict

describe(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

Parameters

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description 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 regressor.

fit(*self*, *X*, *y=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

static load(*file_path*)

Loads component at file path.

Parameters **file_path** (*str*) – Location to load file.

Returns ComponentBase object

needs_fitting(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need 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 XGBoost regressor.

Parameters **X** (*pd.DataFrame*) – Data of shape [n_samples, n_features].

Returns Predicted values.

Return type pd.Series

predict_proba(*self*, *X*)

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.

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.
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_feature_importance	Generate a bar graph of the pipeline's feature importance.
graph_json	Generates a JSON with nodes consisting of the component names and parameters, and edges detailing component relationships.
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.

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_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.

graph_json(*self*)

Generates a JSON with nodes consisting of the component names and parameters, and edges detailing component relationships.

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 serialized JSON representation of a DAG structure.

Return type dag_json (str)

inverse_transform(*self*, *y*)

Apply component inverse_transform methods to estimator predictions in reverse order.

Components that implement inverse_transform are PolynomialDetrender, 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.
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_feature_importance	Generate a bar graph of the pipeline's feature importance.
graph_json	Generates a JSON with nodes consisting of the component names and parameters, and edges detailing component relationships.
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.

`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_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.

`graph_json`(*self*)

Generates a JSON with nodes consisting of the component names and parameters, and edges detailing component relationships.

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 serialized JSON representation of a DAG structure.

Return type dag_json (str)

`inverse_transform`(*self*, *y*)

Apply component inverse_transform methods to estimator predictions in reverse order.

Components that implement inverse_transform are PolynomialDetrender, 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']
...
...
```

(continues on next page)

(continued from previous page)

```
>>> 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',
...                 'categorical_fill_value': None,
...                 'numeric_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.
generate_order	Regenerated the topologically sorted order of the graph.
get_component	Retrieves a single component object from the graph.
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.
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.
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

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)

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_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.

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 PolynomialDetrender, LabelEncoder (tbd).

Parameters **y** – (pd.Series): Final component features.

Returns The target with inverse transformation applied.

Return type pd.Series

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_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.
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_feature_importance	Generate a bar graph of the pipeline's feature importance.
graph_json	Generates a JSON with nodes consisting of the component names and parameters, and edges detailing component relationships.
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.

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_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.

graph_json(*self*)

Generates a JSON with nodes consisting of the component names and parameters, and edges detailing component relationships.

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 serialized JSON representation of a DAG structure.

Return type dag_json (str)

inverse_transform(*self*, *y*)

Apply component inverse_transform methods to estimator predictions in reverse order.

Components that implement inverse_transform are PolynomialDetrender, 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.
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_feature_importance	Generate a bar graph of the pipeline's feature importance.
graph_json	Generates a JSON with nodes consisting of the component names and parameters, and edges detailing component relationships.
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

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_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.

graph_json(*self*)

Generates a JSON with nodes consisting of the component names and parameters, and edges detailing component relationships.

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 serialized JSON representation of a DAG structure.

Return type dag_json (str)

inverse_transform(*self*, *y*)

Apply component inverse_transform methods to estimator predictions in reverse order.

Components that implement inverse_transform are PolynomialDetrender, 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’]
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={"Linear Regressor": {"normalize":␣
↪True}},
...                               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': 'most_frequent', 'fill_value': None},
...     'Linear Regressor': {'fit_intercept': True, 'normalize': 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.
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_feature_importance	Generate a bar graph of the pipeline's feature importance.
graph_json	Generates a JSON with nodes consisting of the component names and parameters, and edges detailing component relationships.
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.

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_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.

graph_json(*self*)

Generates a JSON with nodes consisting of the component names and parameters, and edges detailing component relationships.

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 serialized JSON representation of a DAG structure.

Return type dag_json (str)

inverse_transform(*self*, *y*)

Apply component inverse_transform methods to estimator predictions in reverse order.

Components that implement inverse_transform are PolynomialDetrender, 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**(*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',
```

(continues on next page)

(continued from previous page)

```
...                                             '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.
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.
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_feature_importance	Generate a bar graph of the pipeline's feature importance.
graph_json	Generates a JSON with nodes consisting of the component names and parameters, and edges detailing component relationships.
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.

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

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_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.

graph_json(*self*)

Generates a JSON with nodes consisting of the component names and parameters, and edges detailing component relationships.

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 serialized JSON representation of a DAG structure.

Return type dag_json (str)

inverse_transform(*self*, *y*)

Apply component inverse_transform methods to estimator predictions in reverse order.

Components that implement inverse_transform are PolynomialDetrender, 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 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*)

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.

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.
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.
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_feature_importance	Generate a bar graph of the pipeline's feature importance.
graph_json	Generates a JSON with nodes consisting of the component names and parameters, and edges detailing component relationships.
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.

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

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_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.

graph_json(*self*)

Generates a JSON with nodes consisting of the component names and parameters, and edges detailing component relationships.

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 serialized JSON representation of a DAG structure.

Return type dag_json (str)

inverse_transform(*self*, *y*)

Apply component inverse_transform methods to estimator predictions in reverse order.

Components that implement inverse_transform are PolynomialDetrender, 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 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*)

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.

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.
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.
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_feature_importance	Generate a bar graph of the pipeline's feature importance.
graph_json	Generates a JSON with nodes consisting of the component names and parameters, and edges detailing component relationships.
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.

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

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_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.

graph_json(*self*)

Generates a JSON with nodes consisting of the component names and parameters, and edges detailing component relationships.

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 serialized JSON representation of a DAG structure.

Return type dag_json (str)

inverse_transform(*self*, *y*)

Apply component inverse_transform methods to estimator predictions in reverse order.

Components that implement inverse_transform are PolynomialDetrender, 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 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*)

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.

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.
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.
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_feature_importance	Generate a bar graph of the pipeline's feature importance.
graph_json	Generates a JSON with nodes consisting of the component names and parameters, and edges detailing component relationships.
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.

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

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_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.

graph_json(*self*)

Generates a JSON with nodes consisting of the component names and parameters, and edges detailing component relationships.

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 serialized JSON representation of a DAG structure.

Return type dag_json (str)

inverse_transform(*self*, *y*)

Apply component inverse_transform methods to estimator predictions in reverse order.

Components that implement inverse_transform are PolynomialDetrender, 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 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 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.

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*)

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.

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={"Linear␣
↪Regressor": {"normalize": True},
...                                                     "pipeline": {
↪"gap": 1, "max_delay": 1, "forecast_horizon": 1, "time_index": "date"}},
...                                         custom_name="My␣
↪TimeSeriesRegression Pipeline")
```

(continues on next page)

(continued from previous page)

```
...
>>> 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': 'most_frequent', 'fill_value': None},
...     'Linear Regressor': {'fit_intercept': True, 'normalize': True, 'n_jobs': -1}
↪,
...     'pipeline': {'gap': 1, 'max_delay': 1, 'forecast_horizon': 1, 'time_index':
↪"date"}}
```

Attributes

problem_type	ProblemTypes.TIME_SERIES_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	Fit a time series pipeline.
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_feature_importance	Generate a bar graph of the pipeline's feature importance.
graph_json	Generates a JSON with nodes consisting of the component names and parameters, and edges detailing component relationships.
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.

`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.

`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_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.

graph_json(*self*)

Generates a JSON with nodes consisting of the component names and parameters, and edges detailing component relationships.

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 serialized JSON representation of a DAG structure.

Return type dag_json (str)

inverse_transform(*self*, *y*)

Apply component inverse_transform methods to estimator predictions in reverse order.

Components that implement inverse_transform are PolynomialDetrender, 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 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 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.

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*)

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.

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.
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

logger	

Contents

evalml.pipelines.utils.**generate_pipeline_code**(*element*)

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.

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.

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.**logger**

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*)

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.

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*)

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

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*)

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()`.

Returns Pipeline which can be used to address data check outputs.

Return type PipelineBase

evalml.pipelines.utils.**make_timeseries_baseline_pipeline**(*problem_type*, *gap*, *forecast_horizon*, *time_index*)

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.

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.

continues on next page

Table 7 – continued from previous page

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.
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.
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.
TimeSeriesMulticlassClassificationPipeline	Pipeline base class for time series multiclass classification problems.
TimeSeriesRegressionPipeline	Pipeline base class for time series regression problems.
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.

continues on next page

Table 7 – continued from previous page

XGBoostClassifier	XGBoost Classifier.
XGBoostRegressor	XGBoost Regressor.

Contents

class evalml.pipelines.**ARIMARegressor**(*time_index=None*, *trend=None*, *start_p=2*, *d=0*, *start_q=2*, *max_p=5*, *max_d=2*, *max_q=5*, *seasonal=True*, *n_jobs=- 1*, *random_seed=0*, *maxiter=10*, ***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.
- **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

hyperparameter_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],}
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.
load	Loads component at file path.
needs_fitting	Returns boolean determining 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.

clone(*self*)

Constructs a new component with the same parameters and random state.

Returns A new instance of this component with identical parameters and random state.

default_parameters(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

Returns Default parameters for this component.

Return type dict

describe(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

Parameters

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description 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 with a length of 1 as feature_importance is not defined for ARIMA regressor.

fit(*self*, *X*, *y=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 X was passed to *fit* but not passed in *predict*.

static load(*file_path*)

Loads component at file path.

Parameters **file_path** (*str*) – Location to load file.

Returns ComponentBase object

needs_fitting(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need 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*, *y=None*)

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*)

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.

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.
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_feature_importance	Generate a bar graph of the pipeline's feature importance.
graph_json	Generates a JSON with nodes consisting of the component names and parameters, and edges detailing component relationships.
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.

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_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.

`graph_json`(*self*)

Generates a JSON with nodes consisting of the component names and parameters, and edges detailing component relationships.

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 serialized JSON representation of a DAG structure.

Return type dag_json (str)

`inverse_transform`(*self*, *y*)

Apply component inverse_transform methods to estimator predictions in reverse order.

Components that implement inverse_transform are PolynomialDetrender, 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

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.
load	Loads component at file path.
needs_fitting	Returns boolean determining 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 probability estimates for labels.
save	Saves component at file path.

clone(*self*)

Constructs a new component with the same parameters and random state.

Returns A new instance of this component with identical parameters and random state.

default_parameters(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

Returns Default parameters for this component.

Return type dict

describe(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

Parameters

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description 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

static load(*file_path*)

Loads component at file path.

Parameters **file_path** (*str*) – Location to load file.

Returns ComponentBase object

needs_fitting(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need 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.DataFrame

predict_proba(*self*, *X*)

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.

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

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 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.
load	Loads component at file path.
needs_fitting	Returns boolean determining 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.

clone(*self*)

Constructs a new component with the same parameters and random state.

Returns A new instance of this component with identical parameters and random state.

default_parameters(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

Returns Default parameters for this component.

Return type dict

describe(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

Parameters

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description 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

static load(*file_path*)

Loads component at file path.

Parameters **file_path** (*str*) – Location to load file.

Returns ComponentBase object

needs_fitting(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need 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 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*)

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.

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.
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_feature_importance	Generate a bar graph of the pipeline's feature importance.
graph_json	Generates a JSON with nodes consisting of the component names and parameters, and edges detailing component relationships.
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.

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_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.

graph_json(*self*)

Generates a JSON with nodes consisting of the component names and parameters, and edges detailing component relationships.

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 serialized JSON representation of a DAG structure.

Return type dag_json (str)

inverse_transform(*self*, *y*)

Apply component inverse_transform methods to estimator predictions in reverse order.

Components that implement inverse_transform are PolynomialDetrender, 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',
...                 'categorical_fill_value': None,
...                 'numeric_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,
```

(continues on next page)

(continued from previous page)

```
...                                     '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.
generate_order	Regenerated the topologically sorted order of the graph.
get_component	Retrieves a single component object from the graph.
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.
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.
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

`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)

`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_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.

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 PolynomialDetrender, LabelEncoder (tbd).

Parameters **y** – (pd.Series): Final component features.

Returns The target with inverse transformation applied.

Return type pd.Series

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

hyperparameter_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.
load	Loads component at file path.
needs_fitting	Returns boolean determining 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.

clone(*self*)

Constructs a new component with the same parameters and random state.

Returns A new instance of this component with identical parameters and random state.

default_parameters(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

Returns Default parameters for this component.

Return type dict

describe(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

Parameters

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description 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.

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 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

static load(*file_path*)

Loads component at file path.

Parameters **file_path** (*str*) – Location to load file.

Returns ComponentBase object

needs_fitting(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need 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 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*)

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.

class evalml.pipelines.**DecisionTreeRegressor**(*criterion='mse'*, *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** (*{"mse", "friedman_mse", "mae", "poisson"}*) – The function to measure the quality of a split. Supported criteria are:
  - "mse" 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
  - "mae" 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”: [“mse”, “friedman_mse”, “mae”], “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.
load	Loads component at file path.
needs_fitting	Returns boolean determining 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.

clone(*self*)

Constructs a new component with the same parameters and random state.

Returns A new instance of this component with identical parameters and random state.

default_parameters(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

Returns Default parameters for this component.

Return type dict

describe(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

Parameters

- **print_name** (*bool, optional*) – whether to print name of component

- **return_dict** (*bool, optional*) – whether to return description 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.

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 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

static load(*file_path*)

Loads component at file path.

Parameters **file_path** (*str*) – Location to load file.

Returns ComponentBase object

needs_fitting(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need 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 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*)

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.

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.

Attributes

hyperparameter_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.
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.

clone(*self*)

Constructs a new component with the same parameters and random state.

Returns A new instance of this component with identical parameters and random state.

default_parameters(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

Returns Default parameters for this component.

Return type dict

describe(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

Parameters

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description 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

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

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.

clone(*self*)

Constructs a new component with the same parameters and random state.

Returns A new instance of this component with identical parameters and random state.

default_parameters(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

Returns Default parameters for this component.

Return type dict

describe(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

Parameters

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description 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)

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

hyperparameter_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.
load	Loads component at file path.
needs_fitting	Returns boolean determining 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.

clone(*self*)

Constructs a new component with the same parameters and random state.

Returns A new instance of this component with identical parameters and random state.

default_parameters(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

Returns Default parameters for this component.

Return type dict

describe(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

Parameters

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description 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

static load(*file_path*)

Loads component at file path.

Parameters **file_path** (*str*) – Location to load file.

Returns ComponentBase object

needs_fitting(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need 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 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*)

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.

class evalml.pipelines.**ElasticNetRegressor**(*alpha=0.0001*, *l1_ratio=0.15*, *max_iter=1000*, *normalize=False*, *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.

- **normalize** (*boolean*) – If True, the regressors will be normalized before regression by subtracting the mean and dividing by the l2-norm. Defaults to False.
- **random_seed** (*int*) – Seed for the random number generator. Defaults to 0.

Attributes

hyperparameter_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.
load	Loads component at file path.
needs_fitting	Returns boolean determining 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.

clone(*self*)

Constructs a new component with the same parameters and random state.

Returns A new instance of this component with identical parameters and random state.

default_parameters(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

Returns Default parameters for this component.

Return type dict

describe(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

Parameters

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description 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*, *y=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

static load(*file_path*)

Loads component at file path.

Parameters **file_path** (*str*) – Location to load file.

Returns ComponentBase object

needs_fitting(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need 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 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*)

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.

class evalml.pipelines.**Estimator**(*parameters=None*, *component_obj=None*, *random_seed=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.
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.

clone(*self*)

Constructs a new component with the same parameters and random state.

Returns A new instance of this component with identical parameters and random state.

default_parameters(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

Returns Default parameters for this component.

Return type dict

describe(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

Parameters

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description 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.

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 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

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*)

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*)

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.

class evalml.pipelines.ExponentialSmoothingRegressor(*trend=None*, *damped_trend=False*, *seasonal=None*, *sp=2*, *n_jobs=- 1*, *random_seed=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

hyperparameter_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.
load	Loads component at file path.
needs_fitting	Returns boolean determining 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.

clone(*self*)

Constructs a new component with the same parameters and random state.

Returns A new instance of this component with identical parameters and random state.

default_parameters(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

Returns Default parameters for this component.

Return type dict

describe(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

Parameters

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description 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 with a length of 1 as feature_importance is not defined for Exponential Smoothing regressor.

fit(*self*, *X*, *y=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.

static load(*file_path*)

Loads component at file path.

Parameters **file_path** (*str*) – Location to load file.

Returns ComponentBase object

needs_fitting(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need 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*, *y=None*)

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*)

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.

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

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.
load	Loads component at file path.
needs_fitting	Returns boolean determining 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.

clone(*self*)

Constructs a new component with the same parameters and random state.

Returns A new instance of this component with identical parameters and random state.

default_parameters(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

Returns Default parameters for this component.

Return type dict

describe(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

Parameters

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description 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.

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 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

static load(*file_path*)

Loads component at file path.

Parameters **file_path** (*str*) – Location to load file.

Returns ComponentBase object

needs_fitting(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need 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 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*)

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.

class evalml.pipelines.**ExtraTreesRegressor**(*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 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

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 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.
load	Loads component at file path.
needs_fitting	Returns boolean determining 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.

clone(*self*)

Constructs a new component with the same parameters and random state.

Returns A new instance of this component with identical parameters and random state.

default_parameters(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

Returns Default parameters for this component.

Return type dict

describe(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

Parameters

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description 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.

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 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

static load(*file_path*)

Loads component at file path.

Parameters **file_path** (*str*) – Location to load file.

Returns ComponentBase object

needs_fitting(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need 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 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*)

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.

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.

clone(*self*)

Constructs a new component with the same parameters and random state.

Returns A new instance of this component with identical parameters and random state.

default_parameters(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

Returns Default parameters for this component.

Return type dict

describe(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

Parameters

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description 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

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

hyperparameter_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.
load	Loads component at file path.
needs_fitting	Returns boolean determining 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.

clone(*self*)

Constructs a new component with the same parameters and random state.

Returns A new instance of this component with identical parameters and random state.

default_parameters(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

Returns Default parameters for this component.

Return type dict

describe(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

Parameters

- **print_name** (*bool, optional*) – whether to print name of component

- **return_dict** (*bool, optional*) – whether to return description 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*, *y=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

static load(*file_path*)

Loads component at file path.

Parameters **file_path** (*str*) – Location to load file.

Returns ComponentBase object

needs_fitting(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need 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 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*)

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.

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

hyperparameter_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.
load	Loads component at file path.
needs_fitting	Returns boolean determining 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.

clone(*self*)

Constructs a new component with the same parameters and random state.

Returns A new instance of this component with identical parameters and random state.

default_parameters(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

Returns Default parameters for this component.

Return type dict

describe(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

Parameters

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description 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.

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

static load(*file_path*)

Loads component at file path.

Parameters **file_path** (*str*) – Location to load file.

Returns ComponentBase object

needs_fitting(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need 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.

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

hyperparameter_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.
load	Loads component at file path.
needs_fitting	Returns boolean determining 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.

clone(*self*)

Constructs a new component with the same parameters and random state.

Returns A new instance of this component with identical parameters and random state.

default_parameters(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

Returns Default parameters for this component.

Return type dict

describe(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

Parameters

- **print_name** (*bool, optional*) – whether to print name of component

- **return_dict** (*bool, optional*) – whether to return description 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.

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

static load(*file_path*)

Loads component at file path.

Parameters **file_path** (*str*) – Location to load file.

Returns ComponentBase object

needs_fitting(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need 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*)

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.

class evalml.pipelines.**LinearRegressor**(*fit_intercept=True*, *normalize=False*, *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.
- **normalize** (*boolean*) – If True, the regressors will be normalized before regression by subtracting the mean and dividing by the l2-norm. This parameter is ignored when fit_intercept is set to False. Defaults to False.
- **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], "normalize": [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.
load	Loads component at file path.
needs_fitting	Returns boolean determining 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.

clone(*self*)

Constructs a new component with the same parameters and random state.

Returns A new instance of this component with identical parameters and random state.

default_parameters(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

Returns Default parameters for this component.

Return type dict

describe(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

Parameters

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description 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*, *y=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

static load(*file_path*)

Loads component at file path.

Parameters **file_path** (*str*) – Location to load file.

Returns ComponentBase object

needs_fitting(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need 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 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*)

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.

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

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.
load	Loads component at file path.
needs_fitting	Returns boolean determining 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.

clone(*self*)

Constructs a new component with the same parameters and random state.

Returns A new instance of this component with identical parameters and random state.

default_parameters(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

Returns Default parameters for this component.

Return type dict

describe(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

Parameters

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description 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*, *y=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

static load(*file_path*)

Loads component at file path.

Parameters **file_path** (*str*) – Location to load file.

Returns ComponentBase object

needs_fitting(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need 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 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*)

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.

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.
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_feature_importance	Generate a bar graph of the pipeline's feature importance.
graph_json	Generates a JSON with nodes consisting of the component names and parameters, and edges detailing component relationships.
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.

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_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.

graph_json(*self*)

Generates a JSON with nodes consisting of the component names and parameters, and edges detailing component relationships.

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 serialized JSON representation of a DAG structure.

Return type dag_json (str)

inverse_transform(*self*, *y*)

Apply component inverse_transform methods to estimator predictions in reverse order.

Components that implement inverse_transform are PolynomialDetrender, 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

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.

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

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

hyperparameter_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.

clone(*self*)

Constructs a new component with the same parameters and random state.

Returns A new instance of this component with identical parameters and random state.

default_parameters(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

Returns Default parameters for this component.

Return type dict

describe(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

Parameters

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description 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

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.
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_feature_importance	Generate a bar graph of the pipeline's feature importance.
graph_json	Generates a JSON with nodes consisting of the component names and parameters, and edges detailing component relationships.
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

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_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.

graph_json(*self*)

Generates a JSON with nodes consisting of the component names and parameters, and edges detailing component relationships.

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 serialized JSON representation of a DAG structure.

Return type dag_json (str)

inverse_transform(*self*, *y*)

Apply component inverse_transform methods to estimator predictions in reverse order.

Components that implement inverse_transform are PolynomialDetrender, 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=None*, *changepoint_prior_scale=0.05*, *seasonality_prior_scale=10*, *holidays_prior_scale=10*, *seasonality_mode='additive'*, *random_seed=0*, *stan_backend='CMDSTANPY'*, ***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/

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.
load	Loads component at file path.
needs_fitting	Returns boolean determining 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.

static build_prophet_df(*X*, *y=None*, *time_index='ds'*)

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*)

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*)

Returns array of 0's with len(1) as feature_importance is not defined for Prophet regressor.

fit(*self*, *X*, *y=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*)

Get parameters for the Prophet regressor.

`static load`(*file_path*)

Loads component at file path.

Parameters **file_path** (*str*) – Location to load file.

Returns ComponentBase object

`needs_fitting`(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need 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*, *y=None*)

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*)

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.

`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

hyperparameter_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.
load	Loads component at file path.
needs_fitting	Returns boolean determining 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.

clone(*self*)

Constructs a new component with the same parameters and random state.

Returns A new instance of this component with identical parameters and random state.

default_parameters(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

Returns Default parameters for this component.

Return type dict

describe(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

Parameters

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description 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.

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 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

static load(*file_path*)

Loads component at file path.

Parameters **file_path** (*str*) – Location to load file.

Returns ComponentBase object

needs_fitting(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need 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 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*)

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.

class evalml.pipelines.**RandomForestRegressor**(*n_estimators=100*, *max_depth=6*, *n_jobs=- 1*, *random_seed=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.
load	Loads component at file path.
needs_fitting	Returns boolean determining 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.

clone(*self*)

Constructs a new component with the same parameters and random state.

Returns A new instance of this component with identical parameters and random state.

default_parameters(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

Returns Default parameters for this component.

Return type dict

describe(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

Parameters

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description 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.

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 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

static load(*file_path*)

Loads component at file path.

Parameters **file_path** (*str*) – Location to load file.

Returns ComponentBase object

needs_fitting(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need 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 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*)

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.

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={"Linear Regressor": {"normalize":␣
↪True}},
...                                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': 'most_frequent', 'fill_value': None},
...     'Linear Regressor': {'fit_intercept': True, 'normalize': 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.
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_feature_importance	Generate a bar graph of the pipeline's feature importance.
graph_json	Generates a JSON with nodes consisting of the component names and parameters, and edges detailing component relationships.
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.

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_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.

graph_json(*self*)

Generates a JSON with nodes consisting of the component names and parameters, and edges detailing component relationships.

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 serialized JSON representation of a DAG structure.

Return type dag_json (str)

inverse_transform(*self*, *y*)

Apply component inverse_transform methods to estimator predictions in reverse order.

Components that implement inverse_transform are PolynomialDetrender, 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 0.5. Defaults to None.
- **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.
- **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 -np.inf.
- **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.

clone(*self*)

Constructs a new component with the same parameters and random state.

Returns A new instance of this component with identical parameters and random state.

default_parameters(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

Returns Default parameters for this component.

Return type dict

describe(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

Parameters

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description 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

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. Defaults to None.
- **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.
- **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 -np.inf.
- **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 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.

clone(*self*)

Constructs a new component with the same parameters and random state.

Returns A new instance of this component with identical parameters and random state.

default_parameters(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

Returns Default parameters for this component.

Return type dict

describe(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

Parameters

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description 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

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

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.

clone(*self*)

Constructs a new component with the same parameters and random state.

Returns A new instance of this component with identical parameters and random state.

default_parameters(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

Returns Default parameters for this component.

Return type dict

`describe`(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

Parameters

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description 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

`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

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"],
...     "Stacked Ensemble": [
...         StackedEnsembleClassifier(n_jobs=1, final_
↪estimator=DecisionTreeClassifier()),
...         "Decision Tree.x",
...         "Decision Tree B.x",
...         "y",
...     ],
... }
...
```

(continues on next page)

(continued from previous page)

```
>>> 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

hyperparameter_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.
load	Loads component at file path.
needs_fitting	Returns boolean determining 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.

`clone`(*self*)

Constructs a new component with the same parameters 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*, *y=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

static load(*file_path*)

Loads component at file path.

Parameters **file_path** (*str*) – Location to load file.

Returns ComponentBase object

needs_fitting(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need 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 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*)

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.

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()),
...         "Random Forest.x",
...         "Random Forest B.x",
...         "y",
...     ],
... }
...
>>> cg = ComponentGraph(component_graph)
```

(continues on next page)

(continued from previous page)

```
>>> 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

hyperparameter_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.
load	Loads component at file path.
needs_fitting	Returns boolean determining 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.

clone(*self*)

Constructs a new component with the same parameters 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*, *y=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

static load(*file_path*)

Loads component at file path.

Parameters **file_path** (*str*) – Location to load file.

Returns ComponentBase object

needs_fitting(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need 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 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*)

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.

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

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 component to 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.

clone(*self*)

Constructs a new component with the same parameters and random state.

Returns A new instance of this component with identical parameters and random state.

default_parameters(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

Returns Default parameters for this component.

Return type dict

describe(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

Parameters

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description 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 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

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

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.
load	Loads component at file path.
needs_fitting	Returns boolean determining 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.

clone(*self*)

Constructs a new component with the same parameters and random state.

Returns A new instance of this component with identical parameters and random state.

default_parameters(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

Returns Default parameters for this component.

Return type dict

describe(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

Parameters

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description 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*, *y=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

static load(*file_path*)

Loads component at file path.

Parameters **file_path** (*str*) – Location to load file.

Returns ComponentBase object

needs_fitting(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need 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 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*)

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.

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

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.
load	Loads component at file path.
needs_fitting	Returns boolean determining 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.

clone(*self*)

Constructs a new component with the same parameters and random state.

Returns A new instance of this component with identical parameters and random state.

default_parameters(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

Returns Default parameters for this component.

Return type dict

describe(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

Parameters

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description 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*, *y=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

static load(*file_path*)

Loads component at file path.

Parameters **file_path** (*str*) – Location to load file.

Returns ComponentBase object

needs_fitting(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need 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 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*)

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.

class evalml.pipelines.**TargetEncoder**(*cols=None*, *smoothing=1.0*, *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.

clone(*self*)

Constructs a new component with the same parameters and random state.

Returns A new instance of this component with identical parameters and random state.

default_parameters(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

Returns Default parameters for this component.

Return type dict

describe(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

Parameters

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description 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

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.
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.
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_feature_importance	Generate a bar graph of the pipeline's feature importance.
graph_json	Generates a JSON with nodes consisting of the component names and parameters, and edges detailing component relationships.
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.

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

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_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.

graph_json(*self*)

Generates a JSON with nodes consisting of the component names and parameters, and edges detailing component relationships.

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 serialized JSON representation of a DAG structure.

Return type dag_json (str)

inverse_transform(*self*, *y*)

Apply component inverse_transform methods to estimator predictions in reverse order.

Components that implement inverse_transform are PolynomialDetrender, 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 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*)

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.

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.
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.
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_feature_importance	Generate a bar graph of the pipeline's feature importance.
graph_json	Generates a JSON with nodes consisting of the component names and parameters, and edges detailing component relationships.
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.

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

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_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.

graph_json(*self*)

Generates a JSON with nodes consisting of the component names and parameters, and edges detailing component relationships.

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 serialized JSON representation of a DAG structure.

Return type dag_json (str)

inverse_transform(*self*, *y*)

Apply component inverse_transform methods to estimator predictions in reverse order.

Components that implement inverse_transform are PolynomialDetrender, 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 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*)

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.

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

hyperparameter_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.

clone(*self*)

Constructs a new component with the same parameters and random state.

Returns A new instance of this component with identical parameters and random state.

default_parameters(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

Returns Default parameters for this component.

Return type dict

describe(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

Parameters

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description 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

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")
>>> 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.
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.
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_feature_importance	Generate a bar graph of the pipeline's feature importance.
graph_json	Generates a JSON with nodes consisting of the component names and parameters, and edges detailing component relationships.
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.

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

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_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.

graph_json(*self*)

Generates a JSON with nodes consisting of the component names and parameters, and edges detailing component relationships.

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 serialized JSON representation of a DAG structure.

Return type dag_json (str)

inverse_transform(*self*, *y*)

Apply component inverse_transform methods to estimator predictions in reverse order.

Components that implement inverse_transform are PolynomialDetrender, 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 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*)

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.

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={"Linear␣
↪Regressor": {"normalize": True},
...                                                             "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': 'most_frequent', 'fill_value': None},
...     'Linear Regressor': {'fit_intercept': True, 'normalize': True, 'n_jobs': -1}
↪,
...     'pipeline': {'gap': 1, 'max_delay': 1, 'forecast_horizon': 1, 'time_index':
↪"date"}}
```

Attributes

problem_type	ProblemTypes.TIME_SERIES_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	Fit a time series pipeline.
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_feature_importance	Generate a bar graph of the pipeline's feature importance.
graph_json	Generates a JSON with nodes consisting of the component names and parameters, and edges detailing component relationships.
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.

`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.

`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_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.

graph_json(*self*)

Generates a JSON with nodes consisting of the component names and parameters, and edges detailing component relationships.

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 serialized JSON representation of a DAG structure.

Return type dag_json (str)

inverse_transform(*self*, *y*)

Apply component inverse_transform methods to estimator predictions in reverse order.

Components that implement inverse_transform are PolynomialDetrender, 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 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 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.

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*)

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.

Returns New transformed features.

Return type pd.DataFrame

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.

clone(*self*)

Constructs a new component with the same parameters and random state.

Returns A new instance of this component with identical parameters and random state.

default_parameters(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

Returns Default parameters for this component.

Return type dict

`describe`(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

Parameters

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description 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.

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

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.
load	Loads component at file path.
needs_fitting	Returns boolean determining 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.

clone(*self*)

Constructs a new component with the same parameters and random state.

Returns A new instance of this component with identical parameters and random state.

default_parameters(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

Returns Default parameters for this component.

Return type dict

describe(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

Parameters

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description 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*, *y=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

static load(*file_path*)

Loads component at file path.

Parameters **file_path** (*str*) – Location to load file.

Returns ComponentBase object

needs_fitting(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need 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 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*)

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.

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

hyperparameter_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.
load	Loads component at file path.
needs_fitting	Returns boolean determining 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.

clone(*self*)

Constructs a new component with the same parameters and random state.

Returns A new instance of this component with identical parameters and random state.

default_parameters(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

Returns Default parameters for this component.

Return type dict

describe(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

Parameters

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description 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*, *y=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

static load(*file_path*)

Loads component at file path.

Parameters **file_path** (*str*) – Location to load file.

Returns ComponentBase object

needs_fitting(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need 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 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*)

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.

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

hyperparameter_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.
load	Loads component at file path.
needs_fitting	Returns boolean determining 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.

clone(*self*)

Constructs a new component with the same parameters and random state.

Returns A new instance of this component with identical parameters and random state.

default_parameters(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

Returns Default parameters for this component.

Return type dict

describe(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

Parameters

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description 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*, *y=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

static load(*file_path*)

Loads component at file path.

Parameters **file_path** (*str*) – Location to load file.

Returns ComponentBase object

needs_fitting(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need 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 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*)

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.

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

hyperparameter_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.
load	Loads component at file path.
needs_fitting	Returns boolean determining 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.

clone(*self*)

Constructs a new component with the same parameters and random state.

Returns A new instance of this component with identical parameters and random state.

default_parameters(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

Returns Default parameters for this component.

Return type dict

describe(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

Parameters

- **print_name** (*bool, optional*) – whether to print name of component

- **return_dict** (*bool, optional*) – whether to return description 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

static load(*file_path*)

Loads component at file path.

Parameters **file_path** (*str*) – Location to load file.

Returns ComponentBase object

needs_fitting(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need 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.

class evalml.pipelines.**XGBoostRegressor**(*eta=0.1*, *max_depth=6*, *min_child_weight=1*, *n_estimators=100*, *random_seed=0*, *n_jobs=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

hyperparameter_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.
load	Loads component at file path.
needs_fitting	Returns boolean determining 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.

clone(*self*)

Constructs a new component with the same parameters and random state.

Returns A new instance of this component with identical parameters and random state.

default_parameters(*cls*)

Returns the default parameters for this component.

Our convention is that Component.default_parameters == Component().parameters.

Returns Default parameters for this component.

Return type dict

describe(*self*, *print_name=False*, *return_dict=False*)

Describe a component and its parameters.

Parameters

- **print_name** (*bool, optional*) – whether to print name of component
- **return_dict** (*bool, optional*) – whether to return description 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 regressor.

fit(*self*, *X*, *y=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

static load(*file_path*)

Loads component at file path.

Parameters **file_path** (*str*) – Location to load file.

Returns ComponentBase object

needs_fitting(*self*)

Returns boolean determining if component needs fitting before calling predict, predict_proba, transform, or feature_importances.

This can be overridden to False for components that do not need 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 XGBoost regressor.

Parameters **X** (*pd.DataFrame*) – Data of shape [n_samples, n_features].

Returns Predicted values.

Return type pd.Series

predict_proba(*self*, *X*)

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.

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=1*, *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*) – Number of time units to forecast. Defaults to 1.
- **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=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

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=1*, *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*) – Number of time units to forecast. Defaults to 1.
- **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=0.2*, *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%).
- **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=0.2*, *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%).
- **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=1*, *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*) – Number of time units to forecast. Defaults to 1.
- **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.
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.

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.
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.

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.
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

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.
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

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.
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.

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.
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

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.
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

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'}}
{'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.
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

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'}}
```

(continues on next page)

(continued from previous page)

```
{'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.
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.

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.
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

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’]
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_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.

Contents

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.

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.
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*)

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.

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.**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*) – Rror 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_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_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

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

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.**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.
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.
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.**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.**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*) – Rrror 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*, *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*, *pipeline_parameters=None*, *custom_hyperparameters=None*, *sampler_method='auto'*, *sampler_balanced_ratio=0.25*, *allow_long_running_models=False*, *_ensembling_split_size=0.2*, *_pipelines_per_batch=5*, *automl_algorithm='default'*, *engine='sequential'*, *verbose=False*)

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.
- **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.
- **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.
- **pipeline_parameters** (*dict*) – A dict of the parameters used to initialize a pipeline with. Keys should consist of the component names and values should specify parameter values

  e.g. pipeline_parameters = { 'Imputer' : { 'numeric_impute_strategy': 'most_frequent' } }

- **custom_hyperparameters** (*dict*) – A dict of the hyperparameter ranges used to iterate over during search. Keys should consist of the component names and values should specify a singular value or skopt.Space.

  e.g. custom_hyperparameters = { 'Imputer' : { 'numeric_impute_strategy': Categorical(['most_frequent', 'median']) } }

- **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.

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*, *show_iteration_plot=True*)

Find the best pipeline for the data set.

Parameters **show_iteration_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.

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*)

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.

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*, ***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.
- ****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

v0.48.0 Mar. 25, 2022

- **Enhancements**
  - Replaced `pipeline_parameters` and `custom_hyperparameters` with `search_parameters` in `AutoMLSearch` #3373
  - 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
- Documentation Changes
- **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
- Documentation Changes
- **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

- Enhancements
- **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
- Testing Changes

Warning: Breaking Changes

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 `PolynomialDetrender` 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

INDEX

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

R

S

T

U

V

W

X