EvalML Documentation

Release 0.5.1

Feature Labs, Inc.

Nov 18, 2019

GETTING STARTED

1 Quick Start 3

Index 59

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

Combined with Featuretools and Compose, EvalML can be used to create end-to-end machine learning solutions for classification and regression problems.

CHAPTER

ONE

QUICK START

[1]:
```
import evalml
```

1.1 Load Data

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

[2]:
```
X, y = evalml.demos.load_breast_cancer()
```

1.2 Configure search

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.

[3]:
```
clf = evalml.AutoClassifier(objective="f1",
                            max_pipelines=5)
```

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, test_
↪size=.2)
```

When we call `.fit()`, the search for the best pipeline will begin.

[5]:
```
clf.fit(X_train, y_train)
```

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

Optimizing for F1. Greater score is better.

Searching up to 5 pipelines.
Possible model types: random_forest, linear_model, xgboost

 XGBoost Classifier w/ One Hot Encod...      0%|          | Elapsed:00:00
 XGBoost Classifier w/ One Hot Encod...     20%|        | Elapsed:00:00
```

(continues on next page)

(continued from previous page)

```
Random Forest Classifier w/ One Hot...    40%|      | Elapsed:00:05
XGBoost Classifier w/ One Hot Encod...    60%|    | Elapsed:00:06
Logistic Regression Classifier w/ O...    80%|   | Elapsed:00:13
Logistic Regression Classifier w/ O...   100%|| Elapsed:00:13

Optimization finished
```

1.3 See Pipeline Rankings

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.

[6]:
```
clf.rankings
```

[6]:
```
   id              pipeline_name     score  high_variance_cv  \
0   2    RFClassificationPipeline  0.973822             False
1   4  LogisticRegressionPipeline  0.971963             False
2   1             XGBoostPipeline  0.970312             False
3   0             XGBoostPipeline  0.959800             False
4   3             XGBoostPipeline  0.957570             False

                                      parameters
0  {'n_estimators': 569, 'max_depth': 22, 'impute...
1  {'penalty': 'l2', 'C': 8.444214828324364, 'imp...
2  {'eta': 0.38438170729269994, 'min_child_weight...
3  {'eta': 0.5928446182250184, 'min_child_weight'...
4  {'eta': 0.5288949197529046, 'min_child_weight'...
```

1.4 Describe pipeline

If we are interested in see more details about the pipeline we can describe it using the `id` from the rankings table

[7]:
```
clf.describe_pipeline(3)
```

```
*********************************************************************************************
* XGBoost Classifier w/ One Hot Encoder + Simple Imputer + RF Classifier Select From␣
↪Model *
*********************************************************************************************

Problem Types: Binary Classification, Multiclass Classification
Model Type: XGBoost Classifier
Objective to Optimize: F1 (greater is better)
Number of features: 10

Pipeline Steps
==============
1. One Hot Encoder
2. Simple Imputer
         * impute_strategy : most_frequent
3. RF Classifier Select From Model
         * percent_features : 0.34402219881309576
         * threshold : -inf
```

(continues on next page)

(continued from previous page)

```
4. XGBoost Classifier
         * eta : 0.5288949197529046
         * max_depth : 6
         * min_child_weight : 6.112401049845392

Training
========
Training for Binary Classification problems.
Total training time (including CV): 0.2 seconds

Cross Validation
----------
                F1  Precision  Recall   AUC  Log Loss   MCC # Training # Testing
0            0.974      0.959   0.974 0.995     0.100 0.930    303.000   152.000
1            0.946      0.967   0.946 0.985     0.147 0.863    303.000   152.000
2            0.952      0.957   0.952 0.987     0.155 0.873    304.000   151.000
mean         0.958      0.961   0.958 0.989     0.134 0.889          -         -
std          0.015      0.005   0.015 0.006     0.030 0.036          -         -
coef of var  0.015      0.005   0.015 0.006     0.222 0.041          -         -
```

1.5 Select Best pipeline

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

[8]:
```
pipeline = clf.best_pipeline
pipeline.score(X_holdout, y_holdout)
```

[8]:
```
(0.951048951048951, {})
```

1.5.1 Install

EvalML is available for Python 3.5+. It can be installed by running the following command.:

```
pip install evaml --extra-index-url https://install.featurelabs.com/<license>/
```

1.5.2 Objective Functions

The **objective function** is what EvalML maximizes (or minimizes) as it completes the pipeline search. As it gets feedback from building pipelines, it tunes the hyperparameters to build an optimized models. Therefore, it is critical to have an objective function that captures the how the model's predictions will be used in a business setting.

List of Available Objective Functions

Most AutoML libraries optimize for generic machine learning objective functions. Frequently, the scores produced by the generic machine learning objective diverage from how the model will be evaluated in the real world.

In EvalML, we can train and optimize the model for a specific problem by optimizing a domain-specific objectives functions or by defining our own custom objective function.

Currently, EvalML has two domain specific objective functions with more being developed. For more information on these objective functions click on the links below.

- Fraud Detection
- Lead Scoring

Build your own objective Functions

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 the 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. Read more here.

1.5.3 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 is over 4x better than using a generic machine learning metric like AUC.

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

Configure "Cost of Fraud"

To optimize the pipelines toward the specific business needs of this model, you can set your 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',
)
```

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

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

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

Number of training examples: 99992

Labels
False    84.82%
True     15.18%
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]:
```
X = X.drop(['datetime', 'expiration_date', 'country', 'region', 'provider'], axis=1)
X_train, X_holdout, y_train, y_holdout = evalml.preprocessing.split_data(X, y, test_
↪size=0.2, random_state=0)

print(X.dtypes)
```

```
card_id               int64
store_id              int64
amount                int64
currency             object
customer_present       bool
lat                 float64
lng                 float64
dtype: object
```

Because the fraud labels are binary, we will use `AutoClassifier`. When we call `.fit()`, the search for the best pipeline will begin.

[5]:
```
clf = evalml.AutoClassifier(objective=fraud_objective,
                            additional_objectives=['auc', 'recall', 'precision'],
                            max_pipelines=5)

clf.fit(X_train, y_train)
```

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

Optimizing for Fraud Cost. Lower score is better.

Searching up to 5 pipelines.
Possible model types: linear_model, random_forest, xgboost

 XGBoost Classifier w/ One Hot Encod...     0%|          | Elapsed:00:24
 XGBoost Classifier w/ One Hot Encod...    20%|        | Elapsed:00:51
 Random Forest Classifier w/ One Hot...    40%|      | Elapsed:02:49
 XGBoost Classifier w/ One Hot Encod...    60%|    | Elapsed:03:15
 Logistic Regression Classifier w/ O...    80%|  | Elapsed:03:47
 Logistic Regression Classifier w/ O...   100%|| Elapsed:03:47

 Optimization finished
```

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 fraud detection objective we defined

[6]:
```
clf.rankings
```

[6]:
```
   id              pipeline_name     score  high_variance_cv  \
0   1          XGBoostPipeline  0.007623             False
1   0          XGBoostPipeline  0.007623             False
2   4  LogisticRegressionPipeline  0.007623             False
3   2    RFClassificationPipeline  0.007623             False
4   3          XGBoostPipeline  0.007623             False

                                      parameters
0  {'eta': 0.38438170729269994, 'min_child_weight...
1  {'eta': 0.5928446182250184, 'min_child_weight'...
2  {'penalty': 'l2', 'C': 8.444214828324364, 'imp...
3  {'n_estimators': 569, 'max_depth': 22, 'impute...
4  {'eta': 0.5288949197529046, 'min_child_weight'...
```

to select the best pipeline we can run

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

Describe pipeline

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

[8]:
```
clf.describe_pipeline(clf.rankings.iloc[0]["id"])
```

```
*****************************************************************************************
* XGBoost Classifier w/ One Hot Encoder + Simple Imputer + RF Classifier Select From␣
↪Model *
*****************************************************************************************

Problem Types: Binary Classification, Multiclass Classification
Model Type: XGBoost Classifier
Objective to Optimize: Fraud Cost (lower is better)
Number of features: 5

Pipeline Steps
==============
1. One Hot Encoder
2. Simple Imputer
         * impute_strategy : median
3. RF Classifier Select From Model
         * percent_features : 0.793807787701838
         * threshold : -inf
4. XGBoost Classifier
         * eta : 0.38438170729269994
         * max_depth : 13
         * min_child_weight : 3.677811458900251

Training
========
```

(continues on next page)

(continued from previous page)

```
Training for Binary Classification problems.
Total training time (including CV): 27.5 seconds

Cross Validation
-----------
              Fraud Cost    AUC  Recall  Precision # Training # Testing
0                  0.008 0.864   0.264      0.152  53328.000 26665.000
1                  0.008 0.862   0.264      0.152  53328.000 26665.000
2                  0.008 0.867   0.264      0.152  53330.000 26663.000
mean               0.008 0.864   0.264      0.152          -         -
std                0.000 0.003   0.000      0.000          -         -
coef of var        0.003 0.003   0.000      0.000          -         -
```

Evaluate on hold out

Finally, we retrain the best pipeline on all of the training data and evaluate on the holdout

[9]:
```
best_pipeline.fit(X_train, y_train)
```

[9]:
```
<evalml.pipelines.classification.xgboost.XGBoostPipeline at 0x139088290>
```

Now, we can score the pipeline on the hold out data using both the fraud cost score and the AUC.

[10]:
```
best_pipeline.score(X_holdout, y_holdout, other_objectives=["auc", fraud_objective])
```

[10]:
```
(0.007626457502689945,
 OrderedDict([('AUC', 0.8691817003558158),
              ('Fraud Cost', 0.007626457502689945)]))
```

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.

[11]:
```
clf_auc = evalml.AutoClassifier(objective='auc',
                                additional_objectives=['recall', 'precision'],
                                max_pipelines=5)

clf_auc.fit(X_train, y_train)
```

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

Optimizing for AUC. Greater score is better.

Searching up to 5 pipelines.
Possible model types: linear_model, random_forest, xgboost

 XGBoost Classifier w/ One Hot Encod...     0%|          | Elapsed:00:28
 XGBoost Classifier w/ One Hot Encod...    20%|        | Elapsed:00:58
 Random Forest Classifier w/ One Hot...    40%|      | Elapsed:03:30
 XGBoost Classifier w/ One Hot Encod...    60%|    | Elapsed:04:00
 Logistic Regression Classifier w/ O...    80%|  | Elapsed:04:34
```

(continues on next page)

(continued from previous page)

```
 Logistic Regression Classifier w/ O...   100%|| Elapsed:04:34

 Optimization finished
```

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

[12]:
```
clf_auc.rankings
```

[12]:
```
   id               pipeline_name     score  high_variance_cv  \
0   2    RFClassificationPipeline  0.873053             False
1   1             XGBoostPipeline  0.867186             False
2   0             XGBoostPipeline  0.852527             False
3   3             XGBoostPipeline  0.847393             False
4   4  LogisticRegressionPipeline  0.831181             False

                                        parameters
0  {'n_estimators': 569, 'max_depth': 22, 'impute...
1  {'eta': 0.38438170729269994, 'min_child_weight...
2  {'eta': 0.5928446182250184, 'min_child_weight'...
3  {'eta': 0.5288949197529046, 'min_child_weight'...
4  {'penalty': 'l2', 'C': 8.444214828324364, 'imp...
```

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

# train on the full training data
best_pipeline_auc.fit(X_train, y_train)
```

[13]:
```
<evalml.pipelines.classification.random_forest.RFClassificationPipeline at␣
↪0x10f4fce90>
```

[14]:
```
# get the fraud score on holdout data
best_pipeline_auc.score(X_holdout, y_holdout,  other_objectives=["auc", fraud_
↪objective])
```

[14]:
```
(0.8745605699827037,
 OrderedDict([('AUC', 0.8745605699827037),
              ('Fraud Cost', 0.03273490785793763)]))
```

When we optimize for AUC, we can see that the AUC score from this pipeline is better than the AUC score from the pipeline optimized for fraud cost. However, the losses due to fraud are over 3% of the total transaction amount when optimized for AUC and under 1% when optimized for fraud cost. As a result, we lose more than 2% of the total transaction amount by not optimizing for fraud cost specifically.

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

1.5.4 Custom Objective Functions

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.

How to Create a Objective Function

To create a custom objective function, we must define 2 functions

- The **"objective function"**: this function takes the predictions, true labels, and any other information about the future and returns a score of how well the model performed.
- The **"decision function"**: this function takes prediction probabilities that were output from the model and a threshold and returns a prediction.

To evaluate a particular model, EvalML automatically finds the best threshold to pass to the decision function to generate predictions and then scores the resulting predictions using the objective function. The score from the objective function determines which set of pipeline hyperparameters EvalML will try next.

To give a concrete example, let's look at how the fraud detection objective function is built.

[1]:
```
from evalml.objectives.objective_base import ObjectiveBase

class FraudCost(ObjectiveBase):
    """Score the percentage of money lost of the total transaction amount process due␣
↪to fraud"""
    name = "Fraud Cost"
    needs_fitting = True
    greater_is_better = False
    uses_extra_columns = True
    fit_needs_proba = True
    score_needs_proba = False

    def __init__(self, retry_percentage=.5, interchange_fee=.02,
                 fraud_payout_percentage=1.0, amount_col='amount', verbose=False):
        """Create instance of FraudCost

        Args:
            retry_percentage (float): what percentage of customers will retry a␣
↪transaction if it
                is declined? Between 0 and 1. Defaults to .5

            interchange_fee (float): how much of each successful transaction do you␣
↪collect?
                Between 0 and 1. Defaults to .02

            fraud_payout_percentage (float):  how percentage of fraud will you be␣
↪unable to collect.
                Between 0 and 1. Defaults to 1.0

            amount_col (str): name of column in data that contains the amount.␣
↪defaults to "amount"
        """
        self.retry_percentage = retry_percentage
        self.interchange_fee = interchange_fee
        self.fraud_payout_percentage = fraud_payout_percentage
        self.amount_col = amount_col
        super().__init__(verbose=verbose)

    def decision_function(self, y_predicted, extra_cols, threshold):
        """Determine if transaction is fraud given predicted probabilities,
            dataframe with transaction amount, and threshold"""

        transformed_probs = (y_predicted * extra_cols[self.amount_col])
```

(continues on next page)

(continued from previous page)

```
        return transformed_probs > threshold

    def objective_function(self, y_predicted, y_true, extra_cols):
        """Calculate amount lost to fraud given predictions, true values, and␣
↪dataframe
            with transaction amount"""

        # extract transaction using the amount columns in users data
        transaction_amount = extra_cols[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
```

1.5.5 Setting up pipeline search

Designing the right machine learning pipeline and picking the best parameters is a time-consuming process that relies on a mix of data science intuition as well as trial and error. EvalML streamlines the process of selecting the best modeling algorithms and parameters, so data scientists can focus their energy where it is most needed.

How it works

EvalML selects and tunes machine learning pipelines built of numerous steps. This includes encoding categorical data, missing value imputation, feature selection, feature scaling, and finally machine learning. As EvalML tunes pipelines, it uses the objective function selected and configured by the user to guide its search.

At each iteration, EvalML uses cross-validation to generate an estimate of the pipeline's performances. If a pipeline has high variance across cross-validation folds, it will provide a warning. In this case, the pipeline may not perform reliably in the future.

EvalML is designed to work well out of the box. However, it provides numerous methods for you to control the search described below.

Selecting problem type

EvalML supports both classification and regression problems. You select your problem type by importing the appropriate class.

[1]:
```
import evalml
```

[2]:
```
evalml.AutoClassifier()
```

[2]:
```
<evalml.models.auto_classifier.AutoClassifier at 0x11d782750>
```

[3]:
```
evalml.AutoRegressor()
```

[3]:
```
<evalml.models.auto_regressor.AutoRegressor at 0x11d792950>
```

Setting the Objective Function

The only required parameter to start searching for pipelines is the objective function. Most domain-specific objective functions require you to specify parameters based on your business assumptions. You can do this before you initialize your pipeline search. For example

[4]:
```
from evalml.objectives import FraudCost

fraud_objective = FraudCost(
    retry_percentage=.5,
    interchange_fee=.02,
    fraud_payout_percentage=.75,
    amount_col='amount'
)

evalml.AutoClassifier(objective=fraud_objective)
```

[4]:
```
<evalml.models.auto_classifier.AutoClassifier at 0x11d7abf50>
```

Evaluate on Additional Objectives

Additional objectives can be scored on during the evaluation process. To add another objective, use the `additional_objectives` parameter in AutoClassifier or AutoRegressor. The results of these additional objectives will then appear in the results of `describe_pipeline`.

[5]:
```
from evalml.objectives import FraudCost

fraud_objective = FraudCost(
    retry_percentage=.5,
    interchange_fee=.02,
    fraud_payout_percentage=.75,
    amount_col='amount'
)

evalml.AutoClassifier(objective='AUC', additional_objectives=[fraud_objective])
```

[5]:
```
<evalml.models.auto_classifier.AutoClassifier at 0x11d7b3e50>
```

Selecting Model Types

By default, all model types are considered. You can control which model types to search with the `model_types` parameters

[6]:
```
clf = evalml.AutoClassifier(objective="f1",
                            model_types=["random_forest"])
```

you can see the possible pipelines that will be searched after initialization

[7]:
```
clf.possible_pipelines
```

[7]:
```
[evalml.pipelines.classification.random_forest.RFClassificationPipeline]
```

you can see a list of all supported models like this

[8]:
```
evalml.list_model_types("binary") # `binary` for binary classification and
↪`multiclass` for multiclass classification
```

[8]:
```
[<ModelTypes.LINEAR_MODEL: 'linear_model'>,
 <ModelTypes.XGBOOST: 'xgboost'>,
 <ModelTypes.RANDOM_FOREST: 'random_forest'>]
```

[9]:
```
evalml.list_model_types("regression")
```

[9]:
```
[<ModelTypes.LINEAR_MODEL: 'linear_model'>,
 <ModelTypes.RANDOM_FOREST: 'random_forest'>]
```

Limiting Search Time

You can limit the search time by specifying a maximum number of pipelines and/or a maximum amount of time. EvalML won't build new pipelines after the maximum time has passed or the maximum number of pipelines have been built. If a limit is not set, then a maximum of 5 pipelines will be built.

The maximum search time can be specified as a integer in seconds or as a string in seconds, minutes, or hours.

[10]:
```
evalml.AutoClassifier(objective="f1",
                      max_time=60)

evalml.AutoClassifier(objective="f1",
                      max_time="1 minute")
```

[10]:
```
<evalml.models.auto_classifier.AutoClassifier at 0x11d74c250>
```

To start, EvalML samples 10 sets of hyperparameters chosen randomly for each possible pipeline. Therefore, we recommend setting `max_pipelines` at least 10 times the number of possible pipelines.

[11]:
```
n_possible_pipelines = len(evalml.AutoClassifier(objective="f1").possible_pipelines)
```

[12]:
```
evalml.AutoClassifier(objective="f1",
                      max_time=60,
                      max_pipelines=n_possible_pipelines*10)
```

[12]:
```
<evalml.models.auto_classifier.AutoClassifier at 0x11d7ce750>
```

Control Cross Validation

EvalML cross-validates each model it tests during its search. By default it uses 3-fold cross-validation. You can optionally provide your own cross-validation method.

[13]:
```
from sklearn.model_selection import StratifiedKFold

clf = evalml.AutoClassifier(objective="f1",
                            cv=StratifiedKFold(5))
```

1.5.6 Exploring search results

After finishing a pipeline search, we can inspect the results. First, let's build a search of 10 different pipelines to explore.

[1]:
```
import evalml

X, y = evalml.demos.load_breast_cancer()

clf = evalml.AutoClassifier(objective="f1",
                            max_pipelines=10)

clf.fit(X, y)
```

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

Optimizing for F1. Greater score is better.

Searching up to 10 pipelines.
Possible model types: xgboost, random_forest, linear_model

 XGBoost Classifier w/ One Hot Encod...     0%|          | Elapsed:00:00
 XGBoost Classifier w/ One Hot Encod...    10%|         | Elapsed:00:00
 Random Forest Classifier w/ One Hot...    20%|        | Elapsed:00:06
 XGBoost Classifier w/ One Hot Encod...    30%|       | Elapsed:00:06
 Logistic Regression Classifier w/ O...    40%|      | Elapsed:00:14
 XGBoost Classifier w/ One Hot Encod...    50%|     | Elapsed:00:14
 Logistic Regression Classifier w/ O...    60%|    | Elapsed:00:21
 XGBoost Classifier w/ One Hot Encod...    70%|   | Elapsed:00:22
 Logistic Regression Classifier w/ O...    80%|  | Elapsed:00:29
 Logistic Regression Classifier w/ O...    90%| | Elapsed:00:37
 Logistic Regression Classifier w/ O...   100%|| Elapsed:00:37

 Optimization finished
```

View Rankings

A summary of all the pipelines built can be returned as a dataframe. It is sorted by score. EvalML knows based on your objective function whether or not high or lower is better.

[2]:
```
clf.rankings
```

[2]:

	id	pipeline_name	score	high_variance_cv	\
0	8	LogisticRegressionPipeline	0.980527	False	
1	6	LogisticRegressionPipeline	0.974853	False	
2	9	LogisticRegressionPipeline	0.974853	False	
3	4	LogisticRegressionPipeline	0.973411	False	
4	1	XGBoostPipeline	0.970626	False	

(continues on next page)

(continued from previous page)

```
5   2    RFClassificationPipeline  0.966846            False
6   5             XGBoostPipeline  0.966592            False
7   0             XGBoostPipeline  0.965192            False
8   7             XGBoostPipeline  0.963913            False
9   3             XGBoostPipeline  0.952237            False

                                        parameters
0  {'penalty': 'l2', 'C': 0.5765626434012575, 'im...
1  {'penalty': 'l2', 'C': 6.239401330891865, 'imp...
2  {'penalty': 'l2', 'C': 8.123565600467177, 'imp...
3  {'penalty': 'l2', 'C': 8.444214828324364, 'imp...
4  {'eta': 0.38438170729269994, 'min_child_weight...
5  {'n_estimators': 569, 'max_depth': 22, 'impute...
6  {'eta': 0.6481718720511973, 'min_child_weight'...
7  {'eta': 0.5928446182250184, 'min_child_weight'...
8  {'eta': 0.9786183422327642, 'min_child_weight'...
9  {'eta': 0.5288949197529046, 'min_child_weight'...
```

Describe Pipeline

Each pipeline is given an `id`. We can get more information about any particular pipeline using that id

[3]:
```
clf.describe_pipeline(0)
```

```
*********************************************************************************************
* XGBoost Classifier w/ One Hot Encoder + Simple Imputer + RF Classifier Select From␣
↪Model *
*********************************************************************************************

Problem Types: Binary Classification, Multiclass Classification
Model Type: XGBoost Classifier
Objective to Optimize: F1 (greater is better)
Number of features: 18

Pipeline Steps
==============
1. One Hot Encoder
2. Simple Imputer
         * impute_strategy : most_frequent
3. RF Classifier Select From Model
         * percent_features : 0.6273280598181127
         * threshold : -inf
4. XGBoost Classifier
         * eta : 0.5928446182250184
         * max_depth : 4
         * min_child_weight : 8.598391737229157

Training
========
Training for Binary Classification problems.
Total training time (including CV): 0.2 seconds

Cross Validation
----------
               F1  Precision  Recall   AUC  Log Loss    MCC # Training # Testing
0           0.950      0.935   0.950 0.985     0.154 0.864    379.000   190.000
```

(continues on next page)

(continued from previous page)

```
1            0.975      0.959   0.975 0.996     0.102 0.933    379.000   190.000
2            0.970      0.991   0.970 0.983     0.137 0.923    380.000   189.000
mean         0.965      0.962   0.965 0.988     0.131 0.907          -         -
std          0.013      0.028   0.013 0.007     0.026 0.037          -         -
coef of var  0.014      0.029   0.014 0.007     0.202 0.041          -         -
```

Get Pipeline

You can get the object for any pipeline as well

[4]:
```
clf.get_pipeline(0)
```

[4]:
```
<evalml.pipelines.classification.xgboost.XGBoostPipeline at 0x135081990>
```

Get best pipeline

If you specifically want to get the best pipeline, there is a convenient access.

[5]:
```
clf.best_pipeline
```

[5]:
```
<evalml.pipelines.classification.logistic_regression.LogisticRegressionPipeline at␣
↪0x1372054d0>
```

Feature Importances

We can get the feature importances of the resulting pipeline

[6]:
```
pipeline = clf.get_pipeline(0)
pipeline.feature_importances
```

[6]:
```
    feature  importance
0        22    0.407441
1         7    0.239457
2        27    0.120609
3        20    0.072031
4        23    0.052818
5         6    0.038344
6         1    0.033962
7        21    0.028949
8         4    0.003987
9        25    0.002403
10        0    0.000000
11        2    0.000000
12        3    0.000000
13       12    0.000000
14       13    0.000000
15       18    0.000000
16       19    0.000000
17       29    0.000000
```

Access raw results

You can also get access to all the underlying data like this

[7]:
```
clf.results
```

[7]:
```
{0: {'id': 0,
  'pipeline_name': 'XGBoostPipeline',
  'parameters': {'eta': 0.5928446182250184,
   'min_child_weight': 8.598391737229157,
   'max_depth': 4,
   'impute_strategy': 'most_frequent',
   'percent_features': 0.6273280598181127},
  'score': 0.9651923054186028,
  'high_variance_cv': False,
  'scores': [0.9504132231404958, 0.9752066115702479, 0.9699570815450643],
  'all_objective_scores': [OrderedDict([('F1', 0.9504132231404958),
                ('Precision', 0.9349593495934959),
                ('Recall', 0.9504132231404958),
                ('AUC', 0.984731920937389),
                ('Log Loss', 0.1536501646237938),
                ('MCC', 0.8644170412909863),
                ('# Training', 379),
                ('# Testing', 190)]),
   OrderedDict([('F1', 0.9752066115702479),
                ('Precision', 0.959349593495935),
                ('Recall', 0.9752066115702479),
                ('AUC', 0.9960350337318026),
                ('Log Loss', 0.10194972519713798),
                ('MCC', 0.9327267201397125),
                ('# Training', 379),
                ('# Testing', 190)]),
   OrderedDict([('F1', 0.9699570815450643),
                ('Precision', 0.9912280701754386),
                ('Recall', 0.9699570815450643),
                ('AUC', 0.983313325330132),
                ('Log Loss', 0.13664108953345075),
                ('MCC', 0.9231826763268304),
                ('# Training', 380),
                ('# Testing', 189)])],
  'training_time': 0.248244047164917},
 1: {'id': 1,
  'pipeline_name': 'XGBoostPipeline',
  'parameters': {'eta': 0.38438170729269994,
   'min_child_weight': 3.677811458900251,
   'max_depth': 13,
   'impute_strategy': 'median',
   'percent_features': 0.793807787701838},
  'score': 0.9706261399583499,
  'high_variance_cv': False,
  'scores': [0.9707112970711297, 0.9709543568464729, 0.9702127659574468],
  'all_objective_scores': [OrderedDict([('F1', 0.9707112970711297),
                ('Precision', 0.9666666666666667),
                ('Recall', 0.9707112970711297),
                ('AUC', 0.9917149958574978),
                ('Log Loss', 0.11573912222489813),
                ('MCC', 0.9211268105467613),
                ('# Training', 379),
                ('# Testing', 190)]),
   OrderedDict([('F1', 0.9709543568464729),
                ('Precision', 0.9590163934426229),
                ('Recall', 0.9709543568464729),
```

(continues on next page)

(continued from previous page)

```
                ('AUC', 0.9969227127470707),
                ('Log Loss', 0.07704140599817037),
                ('MCC', 0.9211492315750531),
                ('# Training', 379),
                ('# Testing', 190)]),
   OrderedDict([('F1', 0.9702127659574468),
                ('Precision', 0.9827586206896551),
                ('Recall', 0.9702127659574468),
                ('AUC', 0.9857142857142858),
                ('Log Loss', 0.12628072744331484),
                ('MCC', 0.9218075091290715),
                ('# Training', 380),
                ('# Testing', 189)])],
  'training_time': 0.29195380210876465},
 2: {'id': 2,
  'pipeline_name': 'RFClassificationPipeline',
  'parameters': {'n_estimators': 569,
   'max_depth': 22,
   'impute_strategy': 'most_frequent',
   'percent_features': 0.8593661614465293},
  'score': 0.9668456397284798,
  'high_variance_cv': False,
  'scores': [0.9508196721311476, 0.979253112033195, 0.970464135021097],
  'all_objective_scores': [OrderedDict([('F1', 0.9508196721311476),
                ('Precision', 0.928),
                ('Recall', 0.9508196721311476),
                ('AUC', 0.9889336016096579),
                ('Log Loss', 0.1388421748025717),
                ('MCC', 0.8647724688764672),
                ('# Training', 379),
                ('# Testing', 190)]),
   OrderedDict([('F1', 0.979253112033195),
                ('Precision', 0.9672131147540983),
                ('Recall', 0.979253112033195),
                ('AUC', 0.9898804592259438),
                ('Log Loss', 0.11232987225229708),
                ('MCC', 0.943843520216036),
                ('# Training', 379),
                ('# Testing', 190)]),
   OrderedDict([('F1', 0.970464135021097),
                ('Precision', 0.9745762711864406),
                ('Recall', 0.97046135021097),
                ('AUC', 0.9906362545018007),
                ('Log Loss', 0.11575295379524118),
                ('MCC', 0.9208800271662652),
                ('# Training', 380),
                ('# Testing', 189)])],
  'training_time': 6.06977105140686},
 3: {'id': 3,
  'pipeline_name': 'XGBoostPipeline',
  'parameters': {'eta': 0.5288949197529046,
   'min_child_weight': 6.112401049845392,
   'max_depth': 6,
   'impute_strategy': 'most_frequent',
   'percent_features': 0.34402219881309576},
  'score': 0.9522372250281359,
  'high_variance_cv': False,
```

(continues on next page)

(continued from previous page)

```
  'scores': [0.9367088607594938, 0.9672131147540983, 0.9527896995708156],
  'all_objective_scores': [OrderedDict([('F1', 0.9367088607594938),
                ('Precision', 0.940677966101695),
                ('Recall', 0.9367088607594938),
                ('AUC', 0.9821872410936205),
                ('Log Loss', 0.16857726289155453),
                ('MCC', 0.8318710075349047),
                ('# Training', 379),
                ('# Testing', 190)]),
   OrderedDict([('F1', 0.9672131147540983),
                ('Precision', 0.944),
                ('Recall', 0.9672131147540983),
                ('AUC', 0.9937270682921056),
                ('Log Loss', 0.10433676971098114),
                ('MCC', 0.9106361866954563),
                ('# Training', 379),
                ('# Testing', 190)]),
   OrderedDict([('F1', 0.9527896995708156),
                ('Precision', 0.9736842105263158),
                ('Recall', 0.9527896995708156),
                ('AUC', 0.9845138055222089),
                ('Log Loss', 0.14270813120701523),
                ('MCC', 0.8783921421654207),
                ('# Training', 380),
                ('# Testing', 189)])],
  'training_time': 0.20792675018310547},
 4: {'id': 4,
  'pipeline_name': 'LogisticRegressionPipeline',
  'parameters': {'penalty': 'l2',
   'C': 8.444214828324364,
   'impute_strategy': 'most_frequent'},
  'score': 0.9734109818152151,
  'high_variance_cv': False,
  'scores': [0.970464135021097, 0.9754098360655737, 0.9743589743589743],
  'all_objective_scores': [OrderedDict([('F1', 0.970464135021097),
                ('Precision', 0.9745762711864406),
                ('Recall', 0.970464135021097),
                ('AUC', 0.9885193514025328),
                ('Log Loss', 0.1943294590819038),
                ('MCC', 0.9215733295732883),
                ('# Training', 379),
                ('# Testing', 190)]),
   OrderedDict([('F1', 0.9754098360655737),
                ('Precision', 0.952),
                ('Recall', 0.9754098360655737),
                ('AUC', 0.9849686353414605),
                ('Log Loss', 0.1533799764176819),
                ('MCC', 0.933568045604951),
                ('# Training', 379),
                ('# Testing', 190)]),
   OrderedDict([('F1', 0.9743589743589743),
                ('Precision', 0.991304347826087),
                ('Recall', 0.9743589743589743),
                ('AUC', 0.990516206482593),
                ('Log Loss', 0.1164316714613053),
                ('MCC', 0.9336637889421326),
                ('# Training', 380),
```

(continues on next page)

(continued from previous page)

```
                ('# Testing', 189)])],
  'training_time': 7.461816072463989},
 5: {'id': 5,
  'pipeline_name': 'XGBoostPipeline',
  'parameters': {'eta': 0.6481718720511973,
   'min_child_weight': 4.314173858564932,
   'max_depth': 6,
   'impute_strategy': 'most_frequent',
   'percent_features': 0.871312026764351},
  'score': 0.966592074666908,
  'high_variance_cv': False,
  'scores': [0.9543568464730291, 0.9752066115702479, 0.9702127659574468],
  'all_objective_scores': [OrderedDict([('F1', 0.9543568464730291),
                ('Precision', 0.9426229508196722),
                ('Recall', 0.9543568464730291),
                ('AUC', 0.9899396378269618),
                ('Log Loss', 0.12702225128151967),
                ('MCC', 0.8757606542930872),
                ('# Training', 379),
                ('# Testing', 190)]),
   OrderedDict([('F1', 0.9752066115702479),
                ('Precision', 0.959349593495935),
                ('Recall', 0.9752066115702479),
                ('AUC', 0.9965676411409634),
                ('Log Loss', 0.0801103590350402),
                ('MCC', 0.9327267201397125),
                ('# Training', 379),
                ('# Testing', 190)]),
   OrderedDict([('F1', 0.9702127659574468),
                ('Precision', 0.9827586206896551),
                ('Recall', 0.9702127659574468),
                ('AUC', 0.9858343337334934),
                ('Log Loss', 0.1270006743029361),
                ('MCC', 0.9218075091290715),
                ('# Training', 380),
                ('# Testing', 189)])],
  'training_time': 0.33750486373901367},
 6: {'id': 6,
  'pipeline_name': 'LogisticRegressionPipeline',
  'parameters': {'penalty': 'l2',
   'C': 6.239401330891865,
   'impute_strategy': 'median'},
  'score': 0.9748529087969783,
  'high_variance_cv': False,
  'scores': [0.9747899159663865, 0.9754098360655737, 0.9743589743589743],
  'all_objective_scores': [OrderedDict([('F1', 0.9747899159663865),
                ('Precision', 0.9747899159663865),
                ('Recall', 0.9747899159663865),
                ('AUC', 0.9889927802106758),
                ('Log Loss', 0.17491241567239438),
                ('MCC', 0.932536394839626),
                ('# Training', 379),
                ('# Testing', 190)]),
   OrderedDict([('F1', 0.9754098360655737),
                ('Precision', 0.952),
                ('Recall', 0.9754098360655737),
                ('AUC', 0.9870990649781038),
```

(continues on next page)

(continued from previous page)

```
                ('Log Loss', 0.13982009938625542),
                ('MCC', 0.933568045604951),
                ('# Training', 379),
                ('# Testing', 190)]),
   OrderedDict([('F1', 0.9743589743589743),
                ('Precision', 0.991304347826087),
                ('Recall', 0.9743589743589743),
                ('AUC', 0.990516206482593),
                ('Log Loss', 0.1109645583402926),
                ('MCC', 0.9336637889421326),
                ('# Training', 380),
                ('# Testing', 189)])],
  'training_time': 7.343135118484497},
 7: {'id': 7,
  'pipeline_name': 'XGBoostPipeline',
  'parameters': {'eta': 0.9786183422327642,
   'min_child_weight': 8.192427077950514,
   'max_depth': 20,
   'impute_strategy': 'median',
   'percent_features': 0.6820907348177707},
  'score': 0.9639126305792973,
  'high_variance_cv': False,
  'scores': [0.9547325102880658, 0.9711934156378601, 0.9658119658119659],
  'all_objective_scores': [OrderedDict([('F1', 0.9547325102880658),
                ('Precision', 0.9354838709677419),
                ('Recall', 0.9547325102880658),
                ('AUC', 0.9853237069475678),
                ('Log Loss', 0.15021697619047605),
                ('MCC', 0.8759603969361893),
                ('# Training', 379),
                ('# Testing', 190)]),
   OrderedDict([('F1', 0.9711934156378601),
                ('Precision', 0.9516129032258065),
                ('Recall', 0.9711934156378601),
                ('AUC', 0.9950289975144987),
                ('Log Loss', 0.10607622409680564),
                ('MCC', 0.9216584956231404),
                ('# Training', 379),
                ('# Testing', 190)]),
   OrderedDict([('F1', 0.9658119658119659),
                ('Precision', 0.9826086956521739),
                ('Recall', 0.9658119658119659),
                ('AUC', 0.9834333733493397),
                ('Log Loss', 0.13131227825704234),
                ('MCC', 0.9112159507396058),
                ('# Training', 380),
                ('# Testing', 189)])],
  'training_time': 0.26775383949279785},
 8: {'id': 8,
  'pipeline_name': 'LogisticRegressionPipeline',
  'parameters': {'penalty': 'l2',
   'C': 0.5765626434012575,
   'impute_strategy': 'mean'},
  'score': 0.9805269796885542,
  'high_variance_cv': False,
  'scores': [0.9874476987447698, 0.9754098360655737, 0.9787234042553192],
  'all_objective_scores': [OrderedDict([('F1', 0.9874476987447698),
```

(continues on next page)

(continued from previous page)

```
                ('Precision', 0.9833333333333333),
                ('Recall', 0.9874476987447698),
                ('AUC', 0.994910640312463),
                ('Log Loss', 0.08726565374201126),
                ('MCC', 0.9662335358054943),
                ('# Training', 379),
                ('# Testing', 190)]),
   OrderedDict([('F1', 0.9754098360655737),
                ('Precision', 0.952),
                ('Recall', 0.9754098360655737),
                ('AUC', 0.9979879275653923),
                ('Log Loss', 0.0764559127800754),
                ('MCC', 0.933568045604951),
                ('# Training', 379),
                ('# Testing', 190)]),
   OrderedDict([('F1', 0.9787234042553192),
                ('Precision', 0.9913793103448276),
                ('Recall', 0.9787234042553192),
                ('AUC', 0.9903961584633854),
                ('Log Loss', 0.09774553003325108),
                ('MCC', 0.9443109474170326),
                ('# Training', 380),
                ('# Testing', 189)])],
  'training_time': 7.57702112197876},
 9: {'id': 9,
  'pipeline_name': 'LogisticRegressionPipeline',
  'parameters': {'penalty': 'l2',
   'C': 8.123565600467177,
   'impute_strategy': 'median'},
  'score': 0.9748529087969783,
  'high_variance_cv': False,
  'scores': [0.9747899159663865, 0.9754098360655737, 0.9743589743589743],
  'all_objective_scores': [OrderedDict([('F1', 0.9747899159663865),
                ('Precision', 0.9747899159663865),
                ('Recall', 0.9747899159663865),
                ('AUC', 0.9886377086045686),
                ('Log Loss', 0.19170510282820305),
                ('MCC', 0.932536394839626),
                ('# Training', 379),
                ('# Testing', 190)]),
   OrderedDict([('F1', 0.9754098360655737),
                ('Precision', 0.952),
                ('Recall', 0.9754098360655737),
                ('AUC', 0.9850869925434962),
                ('Log Loss', 0.15159254810085362),
                ('MCC', 0.933568045604951),
                ('# Training', 379),
                ('# Testing', 190)]),
   OrderedDict([('F1', 0.9743589743589743),
                ('Precision', 0.991304347826087),
                ('Recall', 0.9743589743589743),
                ('AUC', 0.990516206482593),
                ('Log Loss', 0.11566930634571038),
                ('MCC', 0.9336637889421326),
                ('# Training', 380),
                ('# Testing', 189)])],
  'training_time': 7.280526161193848}}
```

1.5.7 Avoiding Overfitting

The ultimate goal of machine learning is to make accurate predictions on unseen data. EvalML aims to help you build a model that will perform as you expect once it is deployed in to the real world.

One of the benefits of using EvalML to build models is that it provides guardrails to ensure you are building pipelines that will perform reliably in the future. This page describes the various ways EvalML helps you avoid overfitting to your data.

[1]:
```
import evalml
```

Detecting Label Leakage

A common problem is having features that include information from your label in your training data. By default, EvalML will provide a warning when it detects this may be the case.

Let's set up a simple example to demonstrate what this looks like

[2]:
```
import pandas as pd

X = pd.DataFrame({
    "leaked_feature": [6, 6, 10, 5, 5, 11, 5, 10, 11, 4],
    "leaked_feature_2": [3, 2.5, 5, 2.5, 3, 5.5, 2, 5, 5.5, 2],
    "valid_feature": [3, 1, 3, 2, 4, 6, 1, 3, 3, 11]
})

y = pd.Series([1, 1, 0, 1, 1, 0, 1, 0, 0, 1])

clf = evalml.AutoClassifier(
    max_pipelines=1,
    model_types=["linear_model"],
)

clf.fit(X, y)
```

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

Optimizing for Precision. Greater score is better.

Searching up to 1 pipelines.
Possible model types: linear_model

WARNING: Possible label leakage: leaked_feature, leaked_feature_2
 Logistic Regression Classifier w/ O...     0%|          | Elapsed:00:07
 Logistic Regression Classifier w/ O...   100%|| Elapsed:00:07

 Optimization finished
```

In the example above, EvalML warned about the input features `leaked_feature` and `leak_feature_2`, which are both very closely correlated with the label we are trying to predict. If you'd like to turn this check off, set `detect_label_leakage=False`.

The second way to find features that may be leaking label information is to look at the top features of the model. As we can see below, the top features in our model are the 2 leaked features.

[3]:
```
best_pipeline = clf.best_pipeline
best_pipeline.feature_importances
```

[3]:
```
            feature  importance
0    leaked_feature   -1.773115
1  leaked_feature_2   -1.731261
2     valid_feature   -0.247665
```

Perform cross-validation for pipeline evaluation

By default, EvalML performs 3-fold cross validation when building pipelines. This means that it evaluates each pipeline 3 times using different sets of data for training and testing. In each trial, the data used for testing has no overlap from the data used for training.

While this is a good baseline approach, you can pass your own cross validation object to be used during modeling. The cross validation object can be any of the CV methods defined in scikit-learn or use a compatible API.

For example, if we wanted to do a time series split:

[4]:
```
from sklearn.model_selection import TimeSeriesSplit

X, y = evalml.demos.load_breast_cancer()

clf = evalml.AutoClassifier(
    cv=TimeSeriesSplit(n_splits=6),
    max_pipelines=1
)

clf.fit(X, y)
```

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

Optimizing for Precision. Greater score is better.

Searching up to 1 pipelines.
Possible model types: linear_model, random_forest, xgboost

 XGBoost Classifier w/ One Hot Encod...      0%|          | Elapsed:00:00
 XGBoost Classifier w/ One Hot Encod...   100%|| Elapsed:00:00

 Optimization finished
```

if we describe the 1 pipeline we built, we can see the scores for each of the 6 splits as determined by the cross-validation object we provided. We can also see the number of training examples per fold increased because we were using `TimeSeriesSplit`

[5]:
```
clf.describe_pipeline(0)
```

```
*********************************************************************************************
* XGBoost Classifier w/ One Hot Encoder + Simple Imputer + RF Classifier Select From␣
↪Model *
*********************************************************************************************

Problem Types: Binary Classification, Multiclass Classification
Model Type: XGBoost Classifier
```

(continues on next page)

(continued from previous page)

```
Objective to Optimize: Precision (greater is better)
Number of features: 18

Pipeline Steps
==============
1. One Hot Encoder
2. Simple Imputer
         * impute_strategy : most_frequent
3. RF Classifier Select From Model
         * percent_features : 0.6273280598181127
         * threshold : -inf
4. XGBoost Classifier
         * eta : 0.5928446182250184
         * max_depth : 4
         * min_child_weight : 8.598391737229157

Training
========
Training for Binary Classification problems.
Total training time (including CV): 0.5 seconds

Cross Validation
----------
             Precision    F1  Recall   AUC  Log Loss   MCC # Training # Testing
0                0.974 0.822   0.822 0.950     0.578 0.650     83.000    81.000
1                1.000 0.988   0.988 1.000     0.163 0.976    164.000    81.000
2                0.981 0.981   0.981 0.968     0.139 0.944    245.000    81.000
3                0.963 0.929   0.929 0.991     0.113 0.774    326.000    81.000
4                0.984 0.960   0.960 0.993     0.147 0.830    407.000    81.000
5                0.983 0.983   0.983 0.998     0.083 0.936    488.000    81.000
mean             0.981 0.944   0.944 0.983     0.204 0.852          -         -
std              0.012 0.064   0.064 0.020     0.186 0.125          -         -
coef of var      0.013 0.067   0.067 0.020     0.909 0.147          -         -
```

Detect unstable pipelines

When we perform cross validation we are trying generate an estimate of pipeline performance. EvalML does this by taking the mean of the score across the folds. If the performance across the folds varies greatly, it is indicative the the estimated value may be unreliable.

To protect the user against this, EvalML checks to see if the pipeline's performance has a variance between the different folds. EvalML triggers a warning if the "coefficient of variance" of the scores (the standard deviation divided by mean) of the pipelines scores exeeds .2.

This warning will appear in the pipeline rankings under `high_variance_cv`.

[6]:
```
clf.rankings
```

[6]:
```
   id    pipeline_name     score  high_variance_cv  \
0   0  XGBoostPipeline  0.980845             False

                                        parameters
0  {'eta': 0.5928446182250184, 'min_child_weight'...
```

Create holdout for model validation

EvalML offers a method to quickly create an holdout validation set. A holdout validation set is data that is not used during the process of optimizing or training the model. You should only use this validation set once you've picked the final model you'd like to use.

Below we create a holdout set of 20% of our data

[7]:
```
X, y = evalml.demos.load_breast_cancer()
X_train, X_holdout, y_train, y_holdout = evalml.preprocessing.split_data(X, y, test_
↪size=.2)
```

[8]:
```
clf = evalml.AutoClassifier(
    objective="recall",
    max_pipelines=3,
    detect_label_leakage=True
)
clf.fit(X_train, y_train)
```

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

Optimizing for Recall. Greater score is better.

Searching up to 3 pipelines.
Possible model types: linear_model, random_forest, xgboost

 XGBoost Classifier w/ One Hot Encod...     0%|          | Elapsed:00:00
 XGBoost Classifier w/ One Hot Encod...    33%|      | Elapsed:00:00
 Random Forest Classifier w/ One Hot...    67%|   | Elapsed:00:06
 Random Forest Classifier w/ One Hot...   100%|| Elapsed:00:06

 Optimization finished
```

then we can retrain the best pipeline on all of our training data and see how it performs compared to the estimate

[9]:
```
pipeline = clf.best_pipeline
pipeline.fit(X_train, y_train)
pipeline.score(X_holdout, y_holdout)
```

[9]:
```
(0.951048951048951, {})
```

1.5.8 Regression Example

[1]:
```
import evalml
from evalml.demos import load_diabetes
from evalml.pipelines import PipelineBase, get_pipelines


X, y = evalml.demos.load_diabetes()

clf = evalml.AutoRegressor(objective="R2", max_pipelines=5)

clf.fit(X, y)
```

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

Optimizing for R2. Greater score is better.

Searching up to 5 pipelines.
Possible model types: linear_model, random_forest

 Random Forest Regressor w/ One Hot ...     0%|          | Elapsed:00:05
 Random Forest Regressor w/ One Hot ...    20%|        | Elapsed:00:09
 Linear Regressor w/ One Hot Encoder...    40%|      | Elapsed:00:09
 Random Forest Regressor w/ One Hot ...    40%|      | Elapsed:00:15
 Random Forest Regressor w/ One Hot ...    80%|  | Elapsed:00:21
 Random Forest Regressor w/ One Hot ...   100%|| Elapsed:00:21

 Optimization finished
```

[2]:
```
clf.rankings
```

[2]:
```
   id             pipeline_name     score  high_variance_cv  \
0   2  LinearRegressionPipeline  0.488703             False
1   0      RFRegressionPipeline  0.422322             False
2   4      RFRegressionPipeline  0.391463             False
3   3      RFRegressionPipeline  0.383134             False
4   1      RFRegressionPipeline  0.381204             False

                                      parameters
0  {'impute_strategy': 'mean', 'normalize': True,...
1  {'n_estimators': 569, 'max_depth': 22, 'impute...
2  {'n_estimators': 715, 'max_depth': 7, 'impute_...
3  {'n_estimators': 609, 'max_depth': 7, 'impute_...
4  {'n_estimators': 369, 'max_depth': 10, 'impute...
```

[3]:
```
clf.best_pipeline
```

[3]:
```
<evalml.pipelines.regression.linear_regression.LinearRegressionPipeline at␣
↪0x1308f16d0>
```

[4]:
```
clf.get_pipeline(0)
```

[4]:
```
<evalml.pipelines.regression.random_forest.RFRegressionPipeline at 0x12d737610>
```

[5]:
```
clf.describe_pipeline(0)
```

```
*********************************************************************************************
* Random Forest Regressor w/ One Hot Encoder + Simple Imputer + RF Regressor Select␣
↪From Model *
*********************************************************************************************

Problem Types: Regression
Model Type: Random Forest
Objective to Optimize: R2 (greater is better)
Number of features: 8

Pipeline Steps
==============
1. One Hot Encoder
```

(continues on next page)

(continued from previous page)

```
2. Simple Imputer
         * impute_strategy : most_frequent
3. RF Regressor Select From Model
         * percent_features : 0.8593661614465293
         * threshold : -inf
4. Random Forest Regressor
         * n_estimators : 569
         * max_depth : 22

Training
========
Training for Regression problems.
Total training time (including CV): 5.6 seconds

Cross Validation
----------
               R2    MAE      MSE  MSLE  MedianAE  MaxError  ExpVariance # Training #␣
↪Testing
0           0.427 46.033 3276.018 0.194    39.699   161.858        0.428    294.000  ␣
↪148.000
1           0.450 48.953 3487.566 0.193    44.344   160.513        0.451    295.000  ␣
↪147.000
2           0.390 47.401 3477.117 0.193    41.297   171.420        0.390    295.000  ␣
↪147.000
mean        0.422 47.462 3413.567 0.193    41.780   164.597        0.423          -  ␣
↪       -
std         0.031  1.461  119.235 0.000     2.360     5.947        0.031          -  ␣
↪       -
coef of var 0.072  0.031    0.035 0.002     0.056     0.036        0.073          -  ␣
↪       -
```

1.5.9 Changelog

Future Releases

- Enhancements
- Fixes
- Changes
- Documentation Changes
- Testing Changes

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 contributing instructions #213

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
- **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

1.5.10 Road Map

There are numerous new features and functionality planned for EvalML, some of which are described below:

- Parallelize and distribute model search over cluster
- Export models to python code
- Ability to warm start from a previous pipeline search
- Instructions for adding your own modeling pipelines for EvalML to tune
- Add additional hyperparameter tuning methods
- Visualizations for understanding model search

1.5.11 API Reference

Demo Datasets

load_fraud	Load credit card fraud dataset.
load_wine	Load wine dataset.
load_breast_cancer	Load breast cancer dataset.
load_diabetes	Load diabetes dataset.

evalml.demos.load_fraud

class evalml.demos.**load_fraud**
Load credit card fraud dataset. Binary classification problem

evalml.demos.load_wine

class evalml.demos.**load_wine**
Load wine dataset. Multiclass problem

evalml.demos.load_breast_cancer

class evalml.demos.**load_breast_cancer**
Load breast cancer dataset. Multiclass problem

evalml.demos.load_diabetes

class evalml.demos.**load_diabetes**
Load diabetes dataset. Regression problem

Preprocessing

load_data	Load features and labels from file(s).
split_data	Splits data into train and test sets.

evalml.preprocessing.load_data

class evalml.preprocessing.**load_data**
Load features and labels from file(s).

Parameters

- **path** (*str*) – path to file(s)
- **index** (*str*) – column for index
- **label** (*str*) – column for labels
- **drop** (*list*) – columns to drop
- **verbose** (*bool*) – whether to print information about features and labels

Returns features and labels

Return type DataFrame, Series

evalml.preprocessing.split_data

class evalml.preprocessing.**split_data**
Splits data into train and test sets.

Parameters

- **X** (*DataFrame*) – features
- **y** (*Series*) – labels
- **regression** (*bool*) – if true, do not use stratified split
- **test_size** (*float*) – percent of train set to holdout for testing
- **random_state** (*int*) – seed for the random number generator

Returns features and labels each split into train and test sets

Return type DataFrame, DataFrame, Series, Series

Models

AutoClassifier	Automatic pipeline search for classification problems
AutoRegressor	Automatic pipeline search for regression problems

evalml.AutoClassifier

class evalml.**AutoClassifier**(*objective=None*, *multiclass=False*, *max_pipelines=None*, *max_time=None*, *model_types=None*, *cv=None*, *tuner=None*, *detect_label_leakage=True*, *start_iteration_callback=None*, *add_result_callback=None*, *additional_objectives=None*, *random_state=0*, *verbose=True*)
Automatic pipeline search for classification problems

Methods

__init__	Automated classifier pipeline search
set_problem_type	If there is an objective either:

evalml.AutoClassifier.__init__

AutoClassifier.**__init__**(*objective=None*, *multiclass=False*, *max_pipelines=None*, *max_time=None*, *model_types=None*, *cv=None*, *tuner=None*, *detect_label_leakage=True*, *start_iteration_callback=None*, *add_result_callback=None*, *additional_objectives=None*, *random_state=0*, *verbose=True*)

Automated classifier pipeline search

Parameters

- **objective** (*Object*) – the objective to optimize
- **multiclass** (*bool*) – If True, expecting multiclass data. By default: False.
- **max_pipelines** (*int*) – Maximum number of pipelines to search. If max_pipelines and max_time is not set, then max_pipelines will default to max_pipelines 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.
- **model_types** (*list*) – The model types to search. By default searches over all model_types. Run evalml.list_model_types("classification") to see options.
- **cv** – cross validation method to use. By default StratifiedKFold
- **tuner** – the tuner class to use. Defaults to scikit-optimize tuner
- **detect_label_leakage** (*bool*) – If True, check input features for label leakage and warn if found. Defaults to true.
- **start_iteration_callback** (*callable*) – function called before each pipeline training iteration. Passed two parameters: pipeline_class, parameters.
- **add_result_callback** (*callable*) – function called after each pipeline training iteration. Passed two parameters: results, trained_pipeline.
- **additional_objectives** (*list*) – Custom set of objectives to score on. Will override default objectives for problem type if not empty.
- **random_state** (*int*) – the random_state
- **verbose** (*boolean*) – If True, turn verbosity on. Defaults to True

evalml.AutoClassifier.set_problem_type

AutoClassifier.**set_problem_type**(*objective*, *multiclass*)

If there is an objective either:

a. Set problem_type to MULTICLASS if objective is only multiclass and multiclass is false

b. Set problem_type to MUTLICLASS if multiclass is true

c. Default to BINARY

evalml.AutoRegressor

class evalml.**AutoRegressor**(*objective=None*, *max_pipelines=None*, *max_time=None*, *model_types=None*, *cv=None*, *tuner=None*, *detect_label_leakage=True*, *start_iteration_callback=None*, *add_result_callback=None*, *additional_objectives=None*, *random_state=0*, *verbose=True*)

Automatic pipeline search for regression problems

Methods

__init__	Automated regressors pipeline search

evalml.AutoRegressor.__init__

AutoRegressor.**__init__**(*objective=None*, *max_pipelines=None*, *max_time=None*, *model_types=None*, *cv=None*, *tuner=None*, *detect_label_leakage=True*, *start_iteration_callback=None*, *add_result_callback=None*, *additional_objectives=None*, *random_state=0*, *verbose=True*)

Automated regressors pipeline search

Parameters

- **objective** (*Object*) – the objective to optimize
- **max_pipelines** (*int*) – Maximum number of pipelines to search. If max_pipelines and max_time is not set, then max_pipelines will default to max_pipelines 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.
- **model_types** (*list*) – The model types to search. By default searches over all model_types. Run evalml.list_model_types("regression") to see options.
- **cv** – cross validation method to use. By default StratifiedKFold
- **tuner** – the tuner class to use. Defaults to scikit-optimize tuner
- **detect_label_leakage** (*bool*) – If True, check input features for label leakage and warn if found. Defaults to true.
- **start_iteration_callback** (*callable*) – function called before each pipeline training iteration. Passed two parameters: pipeline_class, parameters.
- **add_result_callback** (*callable*) – function called after each pipeline training iteration. Passed two parameters: results, trained_pipeline.
- **additional_objectives** (*list*) – Custom set of objectives to score on. Will override default objectives for problem type if not empty.
- **random_state** (*int*) – the random_state
- **verbose** (*boolean*) – If True, turn verbosity on. Defaults to True

Model Types

list_model_types	List model type for a particular problem type

evalml.list_model_types

class evalml.**list_model_types**
List model type for a particular problem type

Parameters **problem_types** (*ProblemType or str*) – binary, multiclass, or regression

Returns model_types, list of model types

Pipelines

get_pipelines	Returns potential pipelines by model type
save_pipeline	Saves pipeline at file path
load_pipeline	Loads pipeline at file path
RFClassificationPipeline	Random Forest Pipeline for both binary and multiclass classification
XGBoostPipeline	XGBoost Pipeline for both binary and multiclass classification
LogisticRegressionPipeline	Logistic Regression Pipeline for both binary and multiclass classification
RFRegressionPipeline	Random Forest Pipeline for regression

evalml.pipelines.get_pipelines

class evalml.pipelines.**get_pipelines**
Returns potential pipelines by model type

Parameters

- **problem_type** (*ProblemTypes or str*) – the problem type the pipelines work for.
- **model_types** (*list[ModelTypes or str]*) – model types to match. if none, return all pipelines

Returns

pipelines, list of all pipeline

evalml.pipelines.save_pipeline

class evalml.pipelines.**save_pipeline**
Saves pipeline at file path

Parameters **file_path** (*str*) – location to save file

Returns None

evalml.pipelines.load_pipeline

class evalml.pipelines.**load_pipeline**
Loads pipeline at file path

Parameters **file_path** (*str*) – location to load file

Returns Pipeline obj

evalml.pipelines.RFClassificationPipeline

class evalml.pipelines.**RFClassificationPipeline**(*objective*, *n_estimators*, *max_depth*, *impute_strategy*, *percent_features*, *number_features*, *n_jobs=-1*, *random_state=0*)
Random Forest Pipeline for both binary and multiclass classification

Methods

__init__	Machine learning pipeline made out of transformers and a estimator.

evalml.pipelines.RFClassificationPipeline.__init__

RFClassificationPipeline.**__init__**(*objective*, *n_estimators*, *max_depth*, *impute_strategy*, *percent_features*, *number_features*, *n_jobs=-1*, *random_state=0*)
Machine learning pipeline made out of transformers and a estimator.

Parameters

- **objective** (*Object*) – the objective to optimize
- **component_list** (*list*) – List of components in order
- **random_state** (*int*) – random seed/state
- **n_jobs** (*int*) – Number of jobs to run in parallel

evalml.pipelines.XGBoostPipeline

class evalml.pipelines.**XGBoostPipeline**(*objective*, *eta*, *min_child_weight*, *max_depth*, *impute_strategy*, *percent_features*, *number_features*, *n_estimators=10*, *n_jobs=-1*, *random_state=0*)
XGBoost Pipeline for both binary and multiclass classification

Methods

__init__	Machine learning pipeline made out of transformers and a estimator.

evalml.pipelines.XGBoostPipeline.__init__

XGBoostPipeline.**__init__**(*objective*, *eta*, *min_child_weight*, *max_depth*, *impute_strategy*, *percent_features*, *number_features*, *n_estimators=10*, *n_jobs=-1*, *random_state=0*)

Machine learning pipeline made out of transformers and a estimator.

Parameters

- **objective** (*Object*) – the objective to optimize
- **component_list** (*list*) – List of components in order
- **random_state** (*int*) – random seed/state
- **n_jobs** (*int*) – Number of jobs to run in parallel

evalml.pipelines.LogisticRegressionPipeline

class evalml.pipelines.**LogisticRegressionPipeline**(*objective*, *penalty*, *C*, *impute_strategy*, *number_features*, *n_jobs=-1*, *random_state=0*)

Logistic Regression Pipeline for both binary and multiclass classification

Methods

__init__	Machine learning pipeline made out of transformers and a estimator.

evalml.pipelines.LogisticRegressionPipeline.__init__

LogisticRegressionPipeline.**__init__**(*objective*, *penalty*, *C*, *impute_strategy*, *number_features*, *n_jobs=-1*, *random_state=0*)

Machine learning pipeline made out of transformers and a estimator.

Parameters

- **objective** (*Object*) – the objective to optimize
- **component_list** (*list*) – List of components in order
- **random_state** (*int*) – random seed/state
- **n_jobs** (*int*) – Number of jobs to run in parallel

evalml.pipelines.RFRegressionPipeline

class evalml.pipelines.**RFRegressionPipeline**(*objective*, *n_estimators*, *max_depth*, *impute_strategy*, *percent_features*, *number_features*, *n_jobs=-1*, *random_state=0*)

Random Forest Pipeline for regression

Methods

__init__	Machine learning pipeline made out of transformers and a estimator.

evalml.pipelines.RFRegressionPipeline.__init__

RFRegressionPipeline.**__init__**(*objective*, *n_estimators*, *max_depth*, *impute_strategy*, *percent_features*, *number_features*, *n_jobs=-1*, *random_state=0*)

Machine learning pipeline made out of transformers and a estimator.

Parameters

- **objective** (*Object*) – the objective to optimize
- **component_list** (*list*) – List of components in order
- **random_state** (*int*) – random seed/state
- **n_jobs** (*int*) – Number of jobs to run in parallel

Objective Functions

Domain Specific

FraudCost	Score the percentage of money lost of the total transaction amount process due to fraud
LeadScoring	Lead scoring

evalml.objectives.FraudCost

class evalml.objectives.**FraudCost**(*retry_percentage=0.5*, *interchange_fee=0.02*, *fraud_payout_percentage=1.0*, *amount_col='amount'*, *verbose=False*)

Score the percentage of money lost of the total transaction amount process due to fraud

Methods

__init__	Create instance of FraudCost
decision_function	Determine if transaction is fraud given predicted probabilities, dataframe with transaction amount, and threshold
objective_function	Calculate amount lost to fraud given predictions, true values, and dataframe with transaction amount

evalml.objectives.FraudCost.__init__

FraudCost.**__init__**(*retry_percentage=0.5*, *interchange_fee=0.02*, *fraud_payout_percentage=1.0*, *amount_col='amount'*, *verbose=False*)

Create instance of FraudCost

Parameters

- **retry_percentage** (*float*) – what percentage of customers will retry a transaction if it is declined? Between 0 and 1. Defaults to .5
- **interchange_fee** (*float*) – how much of each successful transaction do you collect? Between 0 and 1. Defaults to .02
- **fraud_payout_percentage** (*float*) – how percentage of fraud will you be unable 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"

evalml.objectives.FraudCost.decision_function

FraudCost.**decision_function**(*y_predicted*, *extra_cols*, *threshold*)

Determine if transaction is fraud given predicted probabilities, dataframe with transaction amount, and threshold

evalml.objectives.FraudCost.objective_function

FraudCost.**objective_function**(*y_predicted*, *y_true*, *extra_cols*)

Calculate amount lost to fraud given predictions, true values, and dataframe with transaction amount

evalml.objectives.LeadScoring

class evalml.objectives.**LeadScoring**(*true_positives=1*, *false_positives=-1*, *verbose=False*)

Lead scoring

Methods

__init__	Create instance.
decision_function	
objective_function	

evalml.objectives.LeadScoring.__init__

LeadScoring.**__init__**(*true_positives=1*, *false_positives=-1*, *verbose=False*)

Create instance.

Parameters

- **label** (*int*) – label to optimize threshold for
- **true_positives** (*int*) – reward for a true positive
- **false_positives** (*int*) – cost for a false positive. Should be negative.

evalml.objectives.LeadScoring.decision_function

LeadScoring.**decision_function**(*y_predicted*, *threshold*)

evalml.objectives.LeadScoring.objective_function

LeadScoring.**objective_function**(*y_predicted*, *y_true*)

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
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
AUC	AUC Score for binary classification
AUCMicro	AUC Score for multiclass classification using micro averaging
AUCMacro	AUC Score for multiclass classification using macro averaging
AUCWeighted	AUC Score for multiclass classification using weighted averaging
LogLoss	Log Loss for both binary and multiclass classification
MCC	Matthews correlation coefficient for both binary and multiclass classification

evalml.objectives.F1

class evalml.objectives.**F1**(*verbose=False*)

F1 Score for binary classification

Methods

score	Calculate score from applying fitted objective to predicted values

evalml.objectives.F1.score

F1.**score**(*y_predicted*, *y_true*)

Calculate score from applying fitted objective to predicted values

If a higher score is better than a lower score, set greater_is_better attribute to True

Parameters

- **y_predicted** (*list*) – the predictions from the model. If needs_proba is True, it is the probability estimates
- **y_true** (*list*) – the ground truth for the predictions.
- **extra_cols** (*extra_cols*) – any extra columns that are needed from training data to fit. Only provided if uses_extra_columns is True.

Returns score

evalml.objectives.F1Micro

class evalml.objectives.**F1Micro**(*verbose=False*)

F1 Score for multiclass classification using micro averaging

Methods

score	Calculate score from applying fitted objective to predicted values

evalml.objectives.F1Micro.score

F1Micro.**score**(*y_predicted*, *y_true*)

Calculate score from applying fitted objective to predicted values

If a higher score is better than a lower score, set greater_is_better attribute to True

Parameters

- **y_predicted** (*list*) – the predictions from the model. If needs_proba is True, it is the probability estimates
- **y_true** (*list*) – the ground truth for the predictions.
- **extra_cols** (*extra_cols*) – any extra columns that are needed from training data to fit. Only provided if uses_extra_columns is True.

Returns score

evalml.objectives.F1Macro

class evalml.objectives.**F1Macro**(*verbose=False*)

F1 Score for multiclass classification using macro averaging

Methods

score	Calculate score from applying fitted objective to predicted values

evalml.objectives.F1Macro.score

F1Macro.**score**(*y_predicted*, *y_true*)

Calculate score from applying fitted objective to predicted values

If a higher score is better than a lower score, set greater_is_better attribute to True

Parameters

- **y_predicted** (*list*) – the predictions from the model. If needs_proba is True, it is the probability estimates
- **y_true** (*list*) – the ground truth for the predictions.
- **extra_cols** (*extra_cols*) – any extra columns that are needed from training data to fit. Only provided if uses_extra_columns is True.

Returns score

evalml.objectives.F1Weighted

class evalml.objectives.**F1Weighted**(*verbose=False*)

F1 Score for multiclass classification using weighted averaging

Methods

score	Calculate score from applying fitted objective to predicted values

evalml.objectives.F1Weighted.score

F1Weighted.**score**(*y_predicted*, *y_true*)

Calculate score from applying fitted objective to predicted values

If a higher score is better than a lower score, set greater_is_better attribute to True

Parameters

- **y_predicted** (*list*) – the predictions from the model. If needs_proba is True, it is the probability estimates
- **y_true** (*list*) – the ground truth for the predictions.
- **extra_cols** (*extra_cols*) – any extra columns that are needed from training data to fit. Only provided if uses_extra_columns is True.

Returns score

evalml.objectives.Precision

class evalml.objectives.**Precision**(*verbose=False*)
Precision Score for binary classification

Methods

score	Calculate score from applying fitted objective to predicted values

evalml.objectives.Precision.score

Precision.**score**(*y_predicted*, *y_true*)
Calculate score from applying fitted objective to predicted values

If a higher score is better than a lower score, set greater_is_better attribute to True

Parameters

- **y_predicted** (*list*) – the predictions from the model. If needs_proba is True, it is the probability estimates
- **y_true** (*list*) – the ground truth for the predictions.
- **extra_cols** (*extra_cols*) – any extra columns that are needed from training data to fit. Only provided if uses_extra_columns is True.

Returns score

evalml.objectives.PrecisionMicro

class evalml.objectives.**PrecisionMicro**(*verbose=False*)
Precision Score for multiclass classification using micro averaging

Methods

score	Calculate score from applying fitted objective to predicted values

evalml.objectives.PrecisionMicro.score

PrecisionMicro.**score**(*y_predicted*, *y_true*)
Calculate score from applying fitted objective to predicted values

If a higher score is better than a lower score, set greater_is_better attribute to True

Parameters

- **y_predicted** (*list*) – the predictions from the model. If needs_proba is True, it is the probability estimates
- **y_true** (*list*) – the ground truth for the predictions.

- **extra_cols** (*extra_cols*) – any extra columns that are needed from training data to fit. Only provided if uses_extra_columns is True.

Returns score

evalml.objectives.PrecisionMacro

class evalml.objectives.**PrecisionMacro**(*verbose=False*)

Precision Score for multiclass classification using macro averaging

Methods

score	Calculate score from applying fitted objective to predicted values

evalml.objectives.PrecisionMacro.score

PrecisionMacro.**score**(*y_predicted*, *y_true*)

Calculate score from applying fitted objective to predicted values

If a higher score is better than a lower score, set greater_is_better attribute to True

Parameters

- **y_predicted** (*list*) – the predictions from the model. If needs_proba is True, it is the probability estimates
- **y_true** (*list*) – the ground truth for the predictions.
- **extra_cols** (*extra_cols*) – any extra columns that are needed from training data to fit. Only provided if uses_extra_columns is True.

Returns score

evalml.objectives.PrecisionWeighted

class evalml.objectives.**PrecisionWeighted**(*verbose=False*)

Precision Score for multiclass classification using weighted averaging

Methods

score	Calculate score from applying fitted objective to predicted values

evalml.objectives.PrecisionWeighted.score

PrecisionWeighted.**score**(*y_predicted*, *y_true*)

Calculate score from applying fitted objective to predicted values

If a higher score is better than a lower score, set greater_is_better attribute to True

Parameters

- **y_predicted** (*list*) – the predictions from the model. If needs_proba is True, it is the probability estimates
- **y_true** (*list*) – the ground truth for the predictions.
- **extra_cols** (*extra_cols*) – any extra columns that are needed from training data to fit. Only provided if uses_extra_columns is True.

Returns score

evalml.objectives.Recall

class evalml.objectives.**Recall**(*verbose=False*)

Recall Score for binary classification

Methods

score	Calculate score from applying fitted objective to predicted values

evalml.objectives.Recall.score

Recall.**score**(*y_predicted*, *y_true*)

Calculate score from applying fitted objective to predicted values

If a higher score is better than a lower score, set greater_is_better attribute to True

Parameters

- **y_predicted** (*list*) – the predictions from the model. If needs_proba is True, it is the probability estimates
- **y_true** (*list*) – the ground truth for the predictions.
- **extra_cols** (*extra_cols*) – any extra columns that are needed from training data to fit. Only provided if uses_extra_columns is True.

Returns score

evalml.objectives.RecallMicro

class evalml.objectives.**RecallMicro**(*verbose=False*)

Recall Score for multiclass classification using micro averaging

Methods

score	Calculate score from applying fitted objective to predicted values

evalml.objectives.RecallMicro.score

RecallMicro.**score**(*y_predicted*, *y_true*)

Calculate score from applying fitted objective to predicted values

If a higher score is better than a lower score, set greater_is_better attribute to True

Parameters

- **y_predicted** (*list*) – the predictions from the model. If needs_proba is True, it is the probability estimates
- **y_true** (*list*) – the ground truth for the predictions.
- **extra_cols** (*extra_cols*) – any extra columns that are needed from training data to fit. Only provided if uses_extra_columns is True.

Returns score

evalml.objectives.RecallMacro

class evalml.objectives.**RecallMacro**(*verbose=False*)

Recall Score for multiclass classification using macro averaging

Methods

score	Calculate score from applying fitted objective to predicted values

evalml.objectives.RecallMacro.score

RecallMacro.**score**(*y_predicted*, *y_true*)

Calculate score from applying fitted objective to predicted values

If a higher score is better than a lower score, set greater_is_better attribute to True

Parameters

- **y_predicted** (*list*) – the predictions from the model. If needs_proba is True, it is the probability estimates
- **y_true** (*list*) – the ground truth for the predictions.
- **extra_cols** (*extra_cols*) – any extra columns that are needed from training data to fit. Only provided if uses_extra_columns is True.

Returns score

evalml.objectives.RecallWeighted

class evalml.objectives.**RecallWeighted**(*verbose=False*)

Recall Score for multiclass classification using weighted averaging

Methods

score	Calculate score from applying fitted objective to predicted values

evalml.objectives.RecallWeighted.score

RecallWeighted.**score**(*y_predicted*, *y_true*)

Calculate score from applying fitted objective to predicted values

If a higher score is better than a lower score, set greater_is_better attribute to True

Parameters

- **y_predicted** (*list*) – the predictions from the model. If needs_proba is True, it is the probability estimates
- **y_true** (*list*) – the ground truth for the predictions.
- **extra_cols** (*extra_cols*) – any extra columns that are needed from training data to fit. Only provided if uses_extra_columns is True.

Returns score

evalml.objectives.AUC

class evalml.objectives.**AUC**(*verbose=False*)

AUC Score for binary classification

Methods

score	Calculate score from applying fitted objective to predicted values

evalml.objectives.AUC.score

AUC.**score**(*y_predicted*, *y_true*)

Calculate score from applying fitted objective to predicted values

If a higher score is better than a lower score, set greater_is_better attribute to True

Parameters

- **y_predicted** (*list*) – the predictions from the model. If needs_proba is True, it is the probability estimates
- **y_true** (*list*) – the ground truth for the predictions.
- **extra_cols** (*extra_cols*) – any extra columns that are needed from training data to fit. Only provided if uses_extra_columns is True.

Returns score

evalml.objectives.AUCMicro

class evalml.objectives.**AUCMicro**(*verbose=False*)
AUC Score for multiclass classification using micro averaging

Methods

score	Calculate score from applying fitted objective to predicted values

evalml.objectives.AUCMicro.score

AUCMicro.**score**(*y_predicted*, *y_true*)
Calculate score from applying fitted objective to predicted values

If a higher score is better than a lower score, set greater_is_better attribute to True

Parameters

- **y_predicted** (*list*) – the predictions from the model. If needs_proba is True, it is the probability estimates
- **y_true** (*list*) – the ground truth for the predictions.
- **extra_cols** (*extra_cols*) – any extra columns that are needed from training data to fit. Only provided if uses_extra_columns is True.

Returns score

evalml.objectives.AUCMacro

class evalml.objectives.**AUCMacro**(*verbose=False*)
AUC Score for multiclass classification using macro averaging

Methods

score	Calculate score from applying fitted objective to predicted values

evalml.objectives.AUCMacro.score

AUCMacro.**score**(*y_predicted*, *y_true*)
Calculate score from applying fitted objective to predicted values

If a higher score is better than a lower score, set greater_is_better attribute to True

Parameters

- **y_predicted** (*list*) – the predictions from the model. If needs_proba is True, it is the probability estimates
- **y_true** (*list*) – the ground truth for the predictions.

- **extra_cols** (*extra_cols*) – any extra columns that are needed from training data to fit. Only provided if uses_extra_columns is True.

Returns score

evalml.objectives.AUCWeighted

class evalml.objectives.**AUCWeighted**(*verbose=False*)

AUC Score for multiclass classification using weighted averaging

Methods

score	Calculate score from applying fitted objective to predicted values

evalml.objectives.AUCWeighted.score

AUCWeighted.**score**(*y_predicted*, *y_true*)

Calculate score from applying fitted objective to predicted values

If a higher score is better than a lower score, set greater_is_better attribute to True

Parameters

- **y_predicted** (*list*) – the predictions from the model. If needs_proba is True, it is the probability estimates
- **y_true** (*list*) – the ground truth for the predictions.
- **extra_cols** (*extra_cols*) – any extra columns that are needed from training data to fit. Only provided if uses_extra_columns is True.

Returns score

evalml.objectives.LogLoss

class evalml.objectives.**LogLoss**(*verbose=False*)

Log Loss for both binary and multiclass classification

Methods

score	Calculate score from applying fitted objective to predicted values

evalml.objectives.LogLoss.score

LogLoss.**score**(*y_predicted*, *y_true*)

Calculate score from applying fitted objective to predicted values

If a higher score is better than a lower score, set greater_is_better attribute to True

Parameters

- **y_predicted** (*list*) – the predictions from the model. If needs_proba is True, it is the probability estimates
- **y_true** (*list*) – the ground truth for the predictions.
- **extra_cols** (*extra_cols*) – any extra columns that are needed from training data to fit. Only provided if uses_extra_columns is True.

Returns score

evalml.objectives.MCC

class evalml.objectives.**MCC**(*verbose=False*)

Matthews correlation coefficient for both binary and multiclass classification

Methods

score	Calculate score from applying fitted objective to predicted values

evalml.objectives.MCC.score

MCC.**score**(*y_predicted*, *y_true*)

Calculate score from applying fitted objective to predicted values

If a higher score is better than a lower score, set greater_is_better attribute to True

Parameters

- **y_predicted** (*list*) – the predictions from the model. If needs_proba is True, it is the probability estimates
- **y_true** (*list*) – the ground truth for the predictions.
- **extra_cols** (*extra_cols*) – any extra columns that are needed from training data to fit. Only provided if uses_extra_columns is True.

Returns score

Regression

R2	Coefficient of determination for regression
MAE	Mean absolute error for regression
MSE	Mean squared error for regression
MSLE	Mean squared log error for regression
MedianAE	Median absolute error for regression
MaxError	Maximum residual error for regression
ExpVariance	Explained variance score for regression

evalml.objectives.R2

class evalml.objectives.**R2**(*verbose=False*)

Coefficient of determination for regression

Methods

score	Calculate score from applying fitted objective to predicted values

evalml.objectives.R2.score

R2.**score**(*y_predicted*, *y_true*)

Calculate score from applying fitted objective to predicted values

If a higher score is better than a lower score, set greater_is_better attribute to True

Parameters

- **y_predicted** (*list*) – the predictions from the model. If needs_proba is True, it is the probability estimates
- **y_true** (*list*) – the ground truth for the predictions.
- **extra_cols** (*extra_cols*) – any extra columns that are needed from training data to fit. Only provided if uses_extra_columns is True.

Returns score

evalml.objectives.MAE

class evalml.objectives.**MAE**(*verbose=False*)

Mean absolute error for regression

Methods

score	Calculate score from applying fitted objective to predicted values

evalml.objectives.MAE.score

MAE.**score**(*y_predicted*, *y_true*)

Calculate score from applying fitted objective to predicted values

If a higher score is better than a lower score, set greater_is_better attribute to True

Parameters

- **y_predicted** (*list*) – the predictions from the model. If needs_proba is True, it is the probability estimates
- **y_true** (*list*) – the ground truth for the predictions.
- **extra_cols** (*extra_cols*) – any extra columns that are needed from training data to fit. Only provided if uses_extra_columns is True.

Returns score

evalml.objectives.MSE

class evalml.objectives.**MSE**(*verbose=False*)
Mean squared error for regression

Methods

score	Calculate score from applying fitted objective to predicted values

evalml.objectives.MSE.score

MSE.**score**(*y_predicted*, *y_true*)
Calculate score from applying fitted objective to predicted values

If a higher score is better than a lower score, set greater_is_better attribute to True

Parameters

- **y_predicted** (*list*) – the predictions from the model. If needs_proba is True, it is the probability estimates
- **y_true** (*list*) – the ground truth for the predictions.
- **extra_cols** (*extra_cols*) – any extra columns that are needed from training data to fit. Only provided if uses_extra_columns is True.

Returns score

evalml.objectives.MSLE

class evalml.objectives.**MSLE**(*verbose=False*)
Mean squared log error for regression

Methods

score	Calculate score from applying fitted objective to predicted values

evalml.objectives.MSLE.score

MSLE.**score**(*y_predicted*, *y_true*)
Calculate score from applying fitted objective to predicted values

If a higher score is better than a lower score, set greater_is_better attribute to True

Parameters

- **y_predicted** (*list*) – the predictions from the model. If needs_proba is True, it is the probability estimates
- **y_true** (*list*) – the ground truth for the predictions.

- **extra_cols** (*extra_cols*) – any extra columns that are needed from training data to fit. Only provided if uses_extra_columns is True.

Returns score

evalml.objectives.MedianAE

class evalml.objectives.**MedianAE**(*verbose=False*)

Median absolute error for regression

Methods

score	Calculate score from applying fitted objective to predicted values

evalml.objectives.MedianAE.score

MedianAE.**score**(*y_predicted*, *y_true*)

Calculate score from applying fitted objective to predicted values

If a higher score is better than a lower score, set greater_is_better attribute to True

Parameters

- **y_predicted** (*list*) – the predictions from the model. If needs_proba is True, it is the probability estimates
- **y_true** (*list*) – the ground truth for the predictions.
- **extra_cols** (*extra_cols*) – any extra columns that are needed from training data to fit. Only provided if uses_extra_columns is True.

Returns score

evalml.objectives.MaxError

class evalml.objectives.**MaxError**(*verbose=False*)

Maximum residual error for regression

Methods

score	Calculate score from applying fitted objective to predicted values

evalml.objectives.MaxError.score

MaxError.**score**(*y_predicted*, *y_true*)

Calculate score from applying fitted objective to predicted values

If a higher score is better than a lower score, set greater_is_better attribute to True

Parameters

- **y_predicted** (*list*) – the predictions from the model. If needs_proba is True, it is the probability estimates
- **y_true** (*list*) – the ground truth for the predictions.
- **extra_cols** (*extra_cols*) – any extra columns that are needed from training data to fit. Only provided if uses_extra_columns is True.

Returns score

evalml.objectives.ExpVariance

class evalml.objectives.**ExpVariance**(*verbose=False*)

Explained variance score for regression

Methods

score	Calculate score from applying fitted objective to predicted values

evalml.objectives.ExpVariance.score

ExpVariance.**score**(*y_predicted*, *y_true*)

Calculate score from applying fitted objective to predicted values

If a higher score is better than a lower score, set greater_is_better attribute to True

Parameters

- **y_predicted** (*list*) – the predictions from the model. If needs_proba is True, it is the probability estimates
- **y_true** (*list*) – the ground truth for the predictions.
- **extra_cols** (*extra_cols*) – any extra columns that are needed from training data to fit. Only provided if uses_extra_columns is True.

Returns score

Problem Types

ProblemTypes	Enum for type of machine learning problem: BINARY, MULTICLASS, or REGRESSION
handle_problem_types	Handles problem_type by either returning the ProblemTypes or converting from a str

evalml.problem_types.ProblemTypes

class evalml.problem_types.**ProblemTypes**

Enum for type of machine learning problem: BINARY, MULTICLASS, or REGRESSION

evalml.problem_types.handle_problem_types

class evalml.problem_types.**handle_problem_types**
Handles problem_type by either returning the ProblemTypes or converting from a str

Parameters **problem_types** (*str or ProblemTypes*) – problem type that needs to be handled

Returns ProblemTypes

Tuners

SKOptTuner	Bayesian Optimizer

evalml.tuners.SKOptTuner

class evalml.tuners.**SKOptTuner**(*space*, *random_state=0*)
Bayesian Optimizer

Methods

__init__	Initialize self.
add	
propose	

evalml.tuners.SKOptTuner.__init__

SKOptTuner.**__init__**(*space*, *random_state=0*)
Initialize self. See help(type(self)) for accurate signature.

evalml.tuners.SKOptTuner.add

SKOptTuner.**add**(*parameters*, *score*)

evalml.tuners.SKOptTuner.propose

SKOptTuner.**propose**()

Guardrails

detect_highly_null	Checks if there are any highly-null columns in a dataframe.
detect_label_leakage	Check if any of the features are highly correlated with the target.

Continued on next page

Table 45 – continued from previous page

detect_outliers	Checks if there are any outliers in a dataframe by using first Isolation Forest to obtain the anomaly score of each index and then using IQR to determine score anomalies.
detect_id_columns	Check if any of the features are ID columns.

evalml.guardrails.detect_highly_null

class evalml.guardrails.**detect_highly_null**

Checks if there are any highly-null columns in a dataframe.

Parameters

- **X** (*DataFrame*) – features
- **percent_threshold** (*float*) – Require that percentage of null values to be considered "highly-null", defaults to .95

Returns A dictionary of features with column name or index and their percentage of null values

evalml.guardrails.detect_label_leakage

class evalml.guardrails.**detect_label_leakage**

Check if any of the features are highly correlated with the target.

Currently only supports binary and numeric targets and features

Parameters

- **X** (*pd.DataFrame*) – The input features to check
- **y** (*pd.Series*) – the labels
- **threshold** (*float*) – the correlation threshold to be considered leakage. Defaults to .95

Returns leakage, dictionary of features with leakage and corresponding threshold

evalml.guardrails.detect_outliers

class evalml.guardrails.**detect_outliers**

Checks if there are any outliers in a dataframe by using first Isolation Forest to obtain the anomaly score of each index and then using IQR to determine score anomalies. Indices with score anomalies are considered outliers.

Parameters **X** (*DataFrame*) – features

Returns A set of indices that may have outlier data.

evalml.guardrails.detect_id_columns

class evalml.guardrails.**detect_id_columns**

Check if any of the features are ID columns. Currently performs these simple checks:

- column name is "id"
- column name ends in "_id"
- column contains all unique values (and is not float / boolean)

Parameters

- **X** (*pd.DataFrame*) – The input features to check
- **threshold** (*float*) – the probability threshold to be considered an ID column. Defaults to 1.0

Returns A dictionary of features with column name or index and their probability of being ID columns

INDEX

Symbols

__init__() (*evalml.AutoClassifier method*), 34
__init__() (*evalml.AutoRegressor method*), 35
__init__() (*evalml.objectives.FraudCost method*), 39
__init__() (*evalml.objectives.LeadScoring method*), 40
__init__() (*evalml.pipelines.LogisticRegressionPipeline method*), 38
__init__() (*evalml.pipelines.RFClassificationPipeline method*), 37
__init__() (*evalml.pipelines.RFRegressionPipeline method*), 39
__init__() (*evalml.pipelines.XGBoostPipeline method*), 38
__init__() (*evalml.tuners.SKOptTuner method*), 56

A

add() (*evalml.tuners.SKOptTuner method*), 56
AUC (*class in evalml.objectives*), 48
AUCMacro (*class in evalml.objectives*), 49
AUCMicro (*class in evalml.objectives*), 49
AUCWeighted (*class in evalml.objectives*), 50
AutoClassifier (*class in evalml*), 33
AutoRegressor (*class in evalml*), 35

D

decision_function() (*evalml.objectives.FraudCost method*), 40
decision_function() (*evalml.objectives.LeadScoring method*), 40
detect_highly_null (*class in evalml.guardrails*), 57
detect_id_columns (*class in evalml.guardrails*), 57
detect_label_leakage (*class in evalml.guardrails*), 57
detect_outliers (*class in evalml.guardrails*), 57

E

ExpVariance (*class in evalml.objectives*), 55

F

F1 (*class in evalml.objectives*), 41
F1Macro (*class in evalml.objectives*), 42
F1Micro (*class in evalml.objectives*), 42
F1Weighted (*class in evalml.objectives*), 43
FraudCost (*class in evalml.objectives*), 39

G

get_pipelines (*class in evalml.pipelines*), 36

H

handle_problem_types (*class in evalml.problem_types*), 56

L

LeadScoring (*class in evalml.objectives*), 40
list_model_types (*class in evalml*), 36
load_breast_cancer (*class in evalml.demos*), 32
load_data (*class in evalml.preprocessing*), 33
load_diabetes (*class in evalml.demos*), 32
load_fraud (*class in evalml.demos*), 32
load_pipeline (*class in evalml.pipelines*), 37
load_wine (*class in evalml.demos*), 32
LogisticRegressionPipeline (*class in evalml.pipelines*), 38
LogLoss (*class in evalml.objectives*), 50

M

MAE (*class in evalml.objectives*), 52
MaxError (*class in evalml.objectives*), 54
MCC (*class in evalml.objectives*), 51
MedianAE (*class in evalml.objectives*), 54
MSE (*class in evalml.objectives*), 53
MSLE (*class in evalml.objectives*), 53

O

objective_function() (*evalml.objectives.FraudCost method*), 40
objective_function() (*evalml.objectives.LeadScoring method*), 41

P

Precision (*class in evalml.objectives*), 44
PrecisionMacro (*class in evalml.objectives*), 45
PrecisionMicro (*class in evalml.objectives*), 44
PrecisionWeighted (*class in evalml.objectives*), 45
ProblemTypes (*class in evalml.problem_types*), 55
propose() (*evalml.tuners.SKOptTuner method*), 56

R

R2 (*class in evalml.objectives*), 51
Recall (*class in evalml.objectives*), 46
RecallMacro (*class in evalml.objectives*), 47
RecallMicro (*class in evalml.objectives*), 46
RecallWeighted (*class in evalml.objectives*), 47
RFClassificationPipeline (*class in evalml.pipelines*), 37
RFRegressionPipeline (*class in evalml.pipelines*), 38

S

save_pipeline (*class in evalml.pipelines*), 36
score() (*evalml.objectives.AUC method*), 48
score() (*evalml.objectives.AUCMacro method*), 49
score() (*evalml.objectives.AUCMicro method*), 49
score() (*evalml.objectives.AUCWeighted method*), 50
score() (*evalml.objectives.ExpVariance method*), 55
score() (*evalml.objectives.F1 method*), 42
score() (*evalml.objectives.F1Macro method*), 43
score() (*evalml.objectives.F1Micro method*), 42
score() (*evalml.objectives.F1Weighted method*), 43
score() (*evalml.objectives.LogLoss method*), 50
score() (*evalml.objectives.MAE method*), 52
score() (*evalml.objectives.MaxError method*), 54
score() (*evalml.objectives.MCC method*), 51
score() (*evalml.objectives.MedianAE method*), 54
score() (*evalml.objectives.MSE method*), 53
score() (*evalml.objectives.MSLE method*), 53
score() (*evalml.objectives.Precision method*), 44
score() (*evalml.objectives.PrecisionMacro method*), 45
score() (*evalml.objectives.PrecisionMicro method*), 44
score() (*evalml.objectives.PrecisionWeighted method*), 45
score() (*evalml.objectives.R2 method*), 52
score() (*evalml.objectives.Recall method*), 46
score() (*evalml.objectives.RecallMacro method*), 47
score() (*evalml.objectives.RecallMicro method*), 47
score() (*evalml.objectives.RecallWeighted method*), 48
set_problem_type() (*evalml.AutoClassifier method*), 34
SKOptTuner (*class in evalml.tuners*), 56
split_data (*class in evalml.preprocessing*), 33

X

XGBoostPipeline (*class in evalml.pipelines*), 37