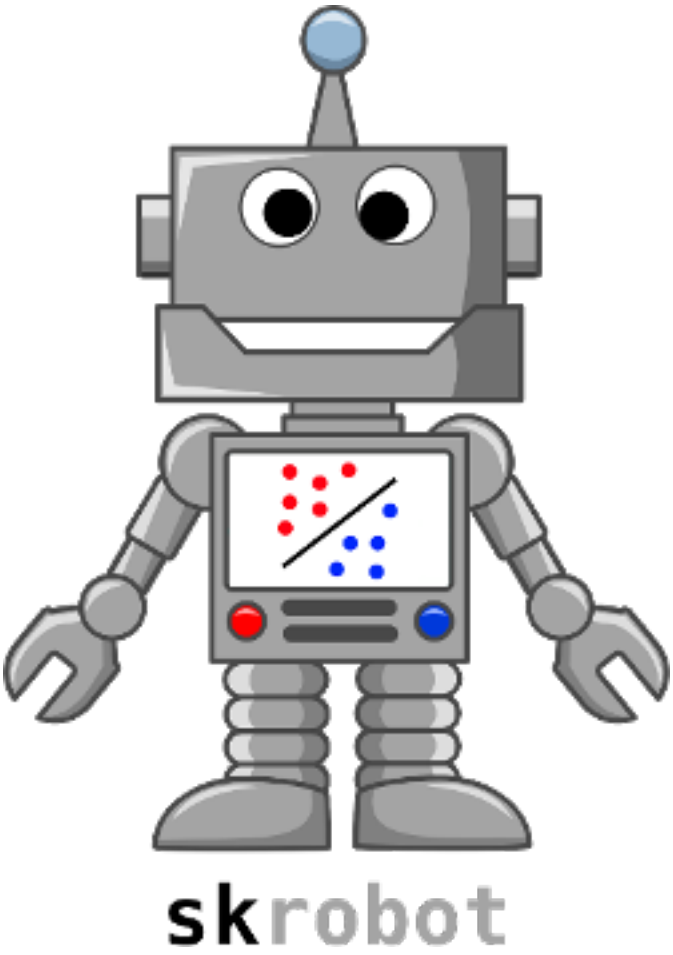
skrobot

Release 1.0.13

Medoid AI

Jan 12, 2021

CONTENTS


skrobot

CHAPTER

ONE

API REFERENCE

1.1 skrobot package

1.1.1 Subpackages

skrobot.core package

Submodules

skrobot.core.experiment module

class skrobot.core.experiment.**Experiment**(*experiments_repository*)
Bases: object

The *Experiment* class can be used to build, track and run an experiment.

It can run *BaseTask* tasks in the context of an experiment.

When building an experiment and/or running tasks, various metadata as well as task-related files are stored for tracking experiments.

Lastly, an experiment can be configured to send notifications when running a task, which can be useful for teams who need to get notified for the progress of the experiment.

__init__(*experiments_repository*)
This is the constructor method and can be used to create a new object instance of *Experiment* class.

Parameters **experiments_repository** (*str*) – The root directory path under which a unique directory is created for the experiment.

set_notifier(*notifier:* skrobot.notification.base_notifier.BaseNotifier)
Optional method.

Set the experiment's notifier.

Parameters **notifier** (*BaseNotifier*) – The experiment's notifier.

Returns The object instance itself.

Return type *Experiment*

set_source_code_file_path(*source_code_file_path*)
Optional method.

Set the experiment's source code file path.

Parameters **source_code_file_path** (*str*) – The experiment's source code file path.

Returns The object instance itself.

Return type *Experiment*

set_experimenter(*experimenter*)
Optional method.

Set the experimenter's name.

By default the experimenter's name is *anonymous*. However, if you want to override it you can pass a new name.

Parameters **experimenter** (*str*) – The experimenter's name.

Returns The object instance itself.

Return type *Experiment*

build()
Build the *Experiment*.

When an experiment is built, it creates a unique directory under which it stores various experiment-related metadata and files for tracking reasons.

Specifically, under the experiment's directory an *experiment.log* JSON file is created, which contains a unique auto-generated experiment ID, the current date & time, and the experimenter's name.

Also, the experiment's directory name contains the experimenter's name as well as current date & time.

Lastly, in case *set_source_code_file_path()* is used, the experiment's source code file is copied also under the experiment's directory.

Returns The object instance itself.

Return type *Experiment*

run(*task*)
Run a *BaseTask* task.

When running a task, its recorded parameters (e.g., *train_task.params*) and any other task-related generated files are stored under experiment's directory for tracking reasons.

The task's recorded parameters are in JSON format.

Also, in case *set_notifier()* is used to set a notifier, a notification is sent for the success or failure (including the error message) of the task's execution.

Lastly, in case an exception occurs, a text file (e.g., *train_task.errors*) is generated under experiment's directory containing the error message.

Parameters **task** (*BaseTask*) – The task to run.

Returns The task's result.

Return type Depends on the `task` parameter.

skrobot.core.task_runner module

class skrobot.core.task_runner.**TaskRunner**(*output_directory_path*)
Bases: object

The *TaskRunner* class is a simplified version (in functionality) of the *Experiment* class.

It leaves out all the "experiment" stuff and is focused mostly in the execution and tracking of *BaseTask* tasks.

__init__(*output_directory_path*)
This is the constructor method and can be used to create a new object instance of *TaskRunner* class.

Parameters **output_directory_path** (*str*) – The output directory path under which task-related generated files are stored.

run(*task*)
Run a *BaseTask* task.

When running a task, its recorded parameters (e.g., *train_task.params*) and any other task-related generated files are stored under output directory for tracking reasons.

The task's recorded parameters are in JSON format.

Lastly, in case an exception occurs, a text file (e.g., *train_task.errors*) is generated under output directory containing the error message.

Parameters **task** (*BaseTask*) – The task to run.

Returns The task's result.

Return type Depends on the task parameter.

skrobot.feature_selection package

Submodules

skrobot.feature_selection.column_selector module

class skrobot.feature_selection.column_selector.**ColumnSelector**(*cols*, *drop_axis=False*)
Bases: sklearn.base.BaseEstimator

The *ColumnSelector* class is an implementation of a column selector for scikit-learn pipelines.

It can be used for manual feature selection to select specific columns from an input data set.

It can select columns either by integer indices or by names.

__init__(*cols*, *drop_axis=False*)
This is the constructor method and can be used to create a new object instance of *ColumnSelector* class.

Parameters

- **cols** (*list*) – A non-empty list specifying the columns to be selected. For example, [1, 4, 5] to select the 2nd, 5th, and 6th columns, and ['A','C','D'] to select the columns A, C and D.
- **drop_axis** (*bool, optional*) – Can be used to reshape the output data set from (n_samples, 1) to (n_samples) by dropping the last axis. It defaults to False.

fit_transform(*X*, *y=None*)
Returns a slice of the input data set.

Parameters

- **X** (*{NumPy array, pandas DataFrame, SciPy sparse matrix}*) – Input vectors of shape (n_samples, n_features), where n_samples is the number of samples and n_features is the number of features.
- **y** (*None*) – Ignored.

Returns Subset of the input data set of shape (n_samples, k_features), where n_samples is the number of samples and k_features <= n_features.

Return type {NumPy array, SciPy sparse matrix}

transform(*X*, *y=None*)
Returns a slice of the input data set.

Parameters

- **X** (*{NumPy array, pandas DataFrame, SciPy sparse matrix}*) – Input vectors of shape (n_samples, n_features), where n_samples is the number of samples and n_features is the number of features.
- **y** (*None*) – Ignored.

Returns Subset of the input data set of shape (n_samples, k_features), where n_samples is the number of samples and k_features <= n_features.

Return type {NumPy array, SciPy sparse matrix}

fit(*X*, *y=None*)
This is a mock method and does nothing.

Parameters

- **X** (*None*) – Ignored.
- **y** (*None*) – Ignored.

Returns The object instance itself.

Return type *ColumnSelector*

skrobot.notification package

Submodules

skrobot.notification.base_notifier module

class skrobot.notification.base_notifier.**BaseNotifier**
Bases: abc.ABC

The *BaseNotifier* is an abstract base class for implementing notifiers.

A notifier can be used to send notifications.

abstract notify(*message*)
An abstract method for sending the notification.

Parameters **message** (*str*) – The notification's message.

skrobot.notification.email_notifier module

class skrobot.notification.email_notifier.**EmailNotifier**(*email_subject*, *sender_account*, *sender_password*, *smtp_server*, *smtp_port*, *recipients*)

Bases: *skrobot.notification.base_notifier.BaseNotifier*

The *EmailNotifier* class can be used to send email notifications.

__init__(*email_subject*, *sender_account*, *sender_password*, *smtp_server*, *smtp_port*, *recipients*)

This is the constructor method and can be used to create a new object instance of *EmailNotifier* class.

Parameters

- **email_subject** (*str*) – The subject of the email.
- **sender_account** (*str*) – The email account of the sender. For example, 'someone@gmail.com'.
- **sender_password** (*str*) – The password of the sender email account.
- **smtp_server** (*str*) – The secured SMTP server of the sender email account. For example, for Gmail is 'smtp.gmail.com'.
- **smtp_port** (*int*) – The port of the secured SMTP server. For example, for Gmail is 465.
- **recipients** (*str*) – The recipients (email addresses) as CSV.

notify(*message*)

Send the email notification.

Parameters **message** (*str*) – The notification's message.

skrobot.tasks package

Submodules

skrobot.tasks.base_task module

class skrobot.tasks.base_task.**BaseTask**(*type_name*, *args*)

Bases: abc.ABC

The *BaseTask* is an abstract base class for implementing tasks.

A task is a configurable and reproducible piece of code built on top of scikit-learn that can be used in machine learning pipelines.

__init__(*type_name*, *args*)

This is the constructor method and can be used from child *BaseTask* implementations.

Parameters

- **type_name** (*str*) – The task's type name. A common practice is to pass the name of the task's class.
- **args** (*dict*) – The task's parameters. A common practice is to pass the parameters at the time of task's object creation. It is a dictionary of key-value pairs, where the key is the parameter name and the value is the parameter value.

get_type()
Get the task's type name.

Returns The task's type name.

Return type str

get_configuration()
Get the task's parameters.

Returns The task's parameters as a dictionary of key-value pairs, where the key is the parameter name and the value is the parameter value.

Return type dict

abstract run(*output_directory*)
An abstract method for running the task.

Parameters **output_directory** (*str*) – The output directory path under which task-related generated files are stored.

skrobot.tasks.base_cross_validation_task module

class skrobot.tasks.base_cross_validation_task.**BaseCrossValidationTask**(*type_name*, *args*)

Bases: *skrobot.tasks.base_task.BaseTask*

The *BaseCrossValidationTask* is an abstract base class for implementing tasks that use cross-validation functionality.

It can support both stratified k-fold cross-validation as well as cross-validation with user-defined folds.

By default, stratified k-fold cross-validation is used with the default parameters of *stratified_folds()* method.

__init__(*type_name*, *args*)
This is the constructor method and can be used from child *BaseCrossValidationTask* implementations.

Parameters

- **type_name** (*str*) – The task's type name. A common practice is to pass the name of the task's class.
- **args** (*dict*) – The task's parameters. A common practice is to pass the parameters at the time of task's object creation. It is a dictionary of key-value pairs, where the key is the parameter name and the value is the parameter value.

custom_folds(*folds_data*, *fold_column='fold'*)
Optional method.

Use cross-validation with user-defined custom folds.

Parameters

- **folds_data** (*{str or pandas DataFrame}*) – The input folds data. It can be either a URL, a disk file path or a pandas DataFrame. The folds data contain the user-defined folds for the samples. If a URL or a disk file path is provided the data must be formatted with the same separation delimiter (comma for CSV, tab for TSV, etc.) as the one used in the input data set files provided to the task. The data must contain two columns and the first row must be the header. The first column is for the sample IDs and needs to

be the same as the one used in the input data set files provided to the task. The second column is for the fold IDs (e.g., 1 through 5, A through D, etc.).

- **fold_column** (*str, optional*) – The column name for the fold IDs. It defaults to 'fold'.

Returns The object instance itself.

Return type *BaseCrossValidationTask*

stratified_folds(*total_folds=3*, *shuffle=False*)
Optional method.

Use stratified k-fold cross-validation.

The folds are made by preserving the percentage of samples for each class.

Parameters

- **total_folds** (*int, optional*) – Number of folds. Must be at least 2. It defaults to 3.
- **shuffle** (*bool, optional*) – Whether to shuffle each class's samples before splitting into batches. Note that the samples within each split will not be shuffled. It defaults to False.

Returns The object instance itself.

Return type *BaseCrossValidationTask*

get_configuration()
Get the task's parameters.

Returns The task's parameters as a dictionary of key-value pairs, where the key is the parameter name and the value is the parameter value.

Return type dict

get_type()
Get the task's type name.

Returns The task's type name.

Return type str

abstract run(*output_directory*)
An abstract method for running the task.

Parameters **output_directory** (*str*) – The output directory path under which task-related generated files are stored.

skrobot.tasks.evaluation_cross_validation_task module

class skrobot.tasks.evaluation_cross_validation_task.**EvaluationCrossValidationTask**(*estimator*, *train_data*, *test_data_s*, *estimator_params*, *field_delim*, *feature_colum*, *id_column*, *label_column*, *random_seed*, *threshold_selecti*, *metric_greater*, *threshold_tuning*, *1.0*, *0.01)*, *export_classi*, *export_confu*, *export_roc_c*, *export_pr_cu*, *export_false_*, *export_false_*, *export_also_*, *fscore_beta*)

Bases: *skrobot.tasks.base_cross_validation_task.BaseCrossValidationTask*

The *EvaluationCrossValidationTask* class can be used to evaluate a scikit-learn estimator/pipeline on some data.

The following evaluation results can be generated on-demand for hold-out test data set as well as train/validation cross-validation folds:

- PR / ROC Curves
- Confusion Matrixes
- Classification Reports
- Performance Metrics

- False Positives
- False Negatives

It can support both stratified k-fold cross-validation as well as cross-validation with user-defined folds.

By default, stratified k-fold cross-validation is used with the default parameters of `stratified_folds()` method.

__init__(*estimator*, *train_data_set*, *test_data_set=None*, *estimator_params=None*, *field_delimiter=','*, *feature_columns='all'*, *id_column='id'*, *label_column='label'*, *random_seed=42*, *threshold_selection_by='f1'*, *metric_greater_is_better=True*, *threshold_tuning_range=(0.01, 1.0, 0.01)*, *export_classification_reports=False*, *export_confusion_matrixes=False*, *export_roc_curves=False*, *export_pr_curves=False*, *export_false_positives_reports=False*, *export_false_negatives_reports=False*, *export_also_for_train_folds=False*, *fscore_beta=1*)

This is the constructor method and can be used to create a new object instance of `EvaluationCrossValidationTask` class.

Parameters

- **estimator** (*scikit-learn {estimator, pipeline}*) – It can be either an estimator (e.g., LogisticRegression) or a pipeline ending with an estimator. The estimator needs to be able to predict probabilities through a `predict_proba` method.
- **train_data_set** (*{str or pandas DataFrame}*) – The input train data set. It can be either a URL, a disk file path or a pandas DataFrame.
- **test_data_set** (*{str or pandas DataFrame}, optional*) – The input test data set. It can be either a URL, a disk file path or a pandas DataFrame. It defaults to None.
- **estimator_params** (*dict, optional*) – The parameters to override in the provided estimator/pipeline. It defaults to None.
- **field_delimiter** (*str, optional*) – The separation delimiter (comma for CSV, tab for TSV, etc.) used in the input train/test data set files. It defaults to ‘,’.
- **feature_columns** (*{str, list}, optional*) – Either ‘all’ to use from the input train/test data set files all the columns or a list of column names to select specific columns. It defaults to ‘all’.
- **id_column** (*str, optional*) – The name of the column in the input train/test data set files containing the sample IDs. It defaults to ‘id’.
- **label_column** (*str, optional*) – The name of the column in the input train/test data set files containing the ground truth labels. It defaults to ‘label’.
- **random_seed** (*int, optional*) – The random seed used in the random number generator. It can be used to reproduce the output. It defaults to 42.
- **threshold_selection_by** (*{str, float}, optional*) – The evaluation results will be generated either for a specific provided threshold value (e.g., 0.49) or for the best threshold found from threshold tuning, based on a specific provided metric (e.g., ‘f1’, ‘f0.55’). It defaults to ‘f1’.
- **metric_greater_is_better** (*bool, optional*) – This flag will control the direction of searching of the best threshold and it depends on the provided metric in `threshold_selection_by`. True, means that greater metric values is better and False means the opposite. It defaults to True.
- **threshold_tuning_range** (*tuple, optional*) – A range in form (start_value, stop_value, step_size) for generating a sequence of threshold values in threshold tuning. It

generates the sequence by incrementing the start value using the step size until it reaches the stop value. It defaults to (0.01, 1.0, 0.01).

- **export_classification_reports** (*bool, optional*) – If this task will export classification reports. It defaults to False.
- **export_confusion_matrixes** (*bool, optional*) – If this task will export confusion matrixes. It defaults to False.
- **export_roc_curves** (*bool, optional*) – If this task will export ROC curves. It defaults to False.
- **export_pr_curves** (*bool, optional*) – If this task will export PR curves. It defaults to False.
- **export_false_positives_reports** (*bool, optional*) – If this task will export false positives reports. It defaults to False.
- **export_false_negatives_reports** (*bool, optional*) – If this task will export false negatives reports. It defaults to False.
- **export_also_for_train_folds** (*bool, optional*) – If this task will export the evaluation results also for the train folds of cross-validation. It defaults to False.
- **fscore_beta** (*float, optional*) – The beta parameter in F-measure. It determines the weight of recall in the score. *beta < 1* lends more weight to precision, while *beta > 1* favors recall (*beta -> 0* considers only precision, *beta -> +inf* only recall). It defaults to 1.

run (*output_directory*)
Run the task.

All of the evaluation results are stored as files under the output directory path.

Parameters **output_directory** (*str*) – The output directory path under which task-related generated files are stored.

Returns The task's result. Specifically, the threshold used along with its related performance metrics and summary metrics from all cross-validation splits as well as hold-out test data set.

Return type dict

custom_folds (*folds_data*, *fold_column='fold'*)
Optional method.

Use cross-validation with user-defined custom folds.

Parameters

- **folds_data** (*{str or pandas DataFrame}*) – The input folds data. It can be either a URL, a disk file path or a pandas DataFrame. The folds data contain the user-defined folds for the samples. If a URL or a disk file path is provided the data must be formatted with the same separation delimiter (comma for CSV, tab for TSV, etc.) as the one used in the input data set files provided to the task. The data must contain two columns and the first row must be the header. The first column is for the sample IDs and needs to be the same as the one used in the input data set files provided to the task. The second column is for the fold IDs (e.g., 1 through 5, A through D, etc.).
- **fold_column** (*str, optional*) – The column name for the fold IDs. It defaults to 'fold'.

Returns The object instance itself.

Return type *BaseCrossValidationTask*

get_configuration()
Get the task's parameters.

Returns The task's parameters as a dictionary of key-value pairs, where the key is the parameter name and the value is the parameter value.

Return type dict

get_type()
Get the task's type name.

Returns The task's type name.

Return type str

stratified_folds(*total_folds=3*, *shuffle=False*)
Optional method.

Use stratified k-fold cross-validation.

The folds are made by preserving the percentage of samples for each class.

Parameters

- **total_folds** (*int, optional*) – Number of folds. Must be at least 2. It defaults to 3.
- **shuffle** (*bool, optional*) – Whether to shuffle each class's samples before splitting into batches. Note that the samples within each split will not be shuffled. It defaults to False.

Returns The object instance itself.

Return type *BaseCrossValidationTask*

skrobot.tasks.feature_selection_cross_validation_task module

class skrobot.tasks.feature_selection_cross_validation_task.**FeatureSelectionCrossValidatio**

Bases: *skrobot.tasks.base_cross_validation_task.BaseCrossValidationTask*

The *FeatureSelectionCrossValidationTask* class can be used to perform feature selection with Recursive Feature Elimination using a scikit-learn estimator on some data.

A scikit-learn preprocessor can be used on the input train data set before feature selection runs.

It can support both stratified k-fold cross-validation as well as cross-validation with user-defined folds.

By default, stratified k-fold cross-validation is used with the default parameters of *stratified_folds()* method.

__init__(*estimator*, *train_data_set*, *estimator_params=None*, *field_delimiter=','*, *preprocessor=None*, *preprocessor_params=None*, *min_features_to_select=1*, *scoring='f1'*, *feature_columns='all'*, *id_column='id'*, *label_column='label'*, *random_seed=42*, *verbose=3*, *n_jobs=1*)

This is the constructor method and can be used to create a new object instance of *FeatureSelectionCrossValidationTask* class.

Parameters

- **estimator** (*scikit-learn estimator*) – An estimator (e.g., LogisticRegression). It needs to provide feature importances through either a `coef_` or a `feature_importances_` attribute.
- **train_data_set** (*{str, pandas DataFrame}*) – The input train data set. It can be either a URL, a disk file path or a pandas DataFrame.

- **estimator_params** (*dict, optional*) – The parameters to override in the provided estimator. It defaults to None.
- **field_delimiter** (*str, optional*) – The separation delimiter (comma for CSV, tab for TSV, etc.) used in the input train data set file. It defaults to ‘,’.
- **preprocessor** (*scikit-learn preprocessor, optional*) – The preprocessor you want to run on the input train data set before feature selection. You can set for example a scikit-learn ColumnTransformer, OneHotEncoder, etc. It defaults to None.
- **preprocessor_params** (*dict, optional*) – The parameters to override in the provided preprocessor. It defaults to None.
- **min_features_to_select** (*int, optional*) – The minimum number of features to be selected. This number of features will always be scored. It defaults to 1.
- **scoring** (*{str, callable}, optional*) – A single scikit-learn scorer string (e.g., ‘f1’) or a callable that is built with scikit-learn `make_scorer`. Note that when using custom scorers, each scorer should return a single value. It defaults to ‘f1’.
- **feature_columns** (*{str, list}, optional*) – Either ‘all’ to use from the input train data set file all the columns or a list of column names to select specific columns. It defaults to ‘all’.
- **id_column** (*str, optional*) – The name of the column in the input train data set file containing the sample IDs. It defaults to ‘id’.
- **label_column** (*str, optional*) – The name of the column in the input train data set file containing the ground truth labels. It defaults to ‘label’.
- **random_seed** (*int, optional*) – The random seed used in the random number generator. It can be used to reproduce the output. It defaults to 42.
- **verbose** (*int, optional*) – Controls the verbosity of output. The higher, the more messages. It defaults to 3.
- **n_jobs** (*int, optional*) – Number of jobs to run in parallel. -1 means using all processors. It defaults to 1.

run (*output_directory*)
Run the task.

The selected features are returned as a result and also stored in a *features_selected.txt* file under the output directory path.

Parameters **output_directory** (*str*) – The output directory path under which task-related generated files are stored.

Returns The task’s result. Specifically, the selected features, which can be either column names from the input train data set or column indexes from the preprocessed data set, depending on whether a `preprocessor` was used or not.

Return type list

custom_folds (*folds_data*, *fold_column='fold'*)
Optional method.

Use cross-validation with user-defined custom folds.

Parameters

- **folds_data** (*{str or pandas DataFrame}*) – The input folds data. It can be either a URL, a disk file path or a pandas DataFrame. The folds data contain the user-defined folds for the samples. If a URL or a disk file path is provided the data must be

formatted with the same separation delimiter (comma for CSV, tab for TSV, etc.) as the one used in the input data set files provided to the task. The data must contain two columns and the first row must be the header. The first column is for the sample IDs and needs to be the same as the one used in the input data set files provided to the task. The second column is for the fold IDs (e.g., 1 through 5, A through D, etc.).

- **fold_column** (*str, optional*) – The column name for the fold IDs. It defaults to 'fold'.

Returns The object instance itself.

Return type *BaseCrossValidationTask*

get_configuration()
Get the task's parameters.

Returns The task's parameters as a dictionary of key-value pairs, where the key is the parameter name and the value is the parameter value.

Return type dict

get_type()
Get the task's type name.

Returns The task's type name.

Return type str

stratified_folds(*total_folds=3*, *shuffle=False*)
Optional method.

Use stratified k-fold cross-validation.

The folds are made by preserving the percentage of samples for each class.

Parameters

- **total_folds** (*int, optional*) – Number of folds. Must be at least 2. It defaults to 3.
- **shuffle** (*bool, optional*) – Whether to shuffle each class's samples before splitting into batches. Note that the samples within each split will not be shuffled. It defaults to False.

Returns The object instance itself.

Return type *BaseCrossValidationTask*

skrobot.tasks.hyperparameters_search_cross_validation_task module

class skrobot.tasks.hyperparameters_search_cross_validation_task.**HyperParametersSearchCross**

Bases: *skrobot.tasks.base_cross_validation_task.BaseCrossValidationTask*

The *HyperParametersSearchCrossValidationTask* class can be used to search the best hyperparameters of a scikit-learn estimator/pipeline on some data.

Cross-Validation

It can support both stratified k-fold cross-validation as well as cross-validation with user-defined folds.

By default, stratified k-fold cross-validation is used with the default parameters of *stratified_folds()* method.

Search

It can support both grid search as well as random search.

By default, grid search is used.

__init__(*estimator*, *search_params*, *train_data_set*, *estimator_params=None*, *field_delimiter=','*, *scorers=['roc_auc', 'average_precision', 'f1', 'precision', 'recall', 'accuracy']*, *feature_columns='all'*, *id_column='id'*, *label_column='label'*, *objective_score='f1'*, *random_seed=42*, *verbose=3*, *n_jobs=1*, *return_train_score=True*)

This is the constructor method and can be used to create a new object instance of *HyperParametersSearchCrossValidationTask* class.

Parameters

- **estimator** (*scikit-learn {estimator, pipeline}*) – It can be either an estimator (e.g., LogisticRegression) or a pipeline ending with an estimator.
- **search_params** (*{dict, list of dictionaries}*) – Dictionary with hyperparameters names as keys and lists of hyperparameter settings to try as values, or a list of such dictionaries, in which case the grids spanned by each dictionary in the list are explored. This enables searching over any sequence of hyperparameter settings.
- **train_data_set** (*{str, pandas DataFrame}*) – The input train data set. It can be either a URL, a disk file path or a pandas DataFrame.
- **estimator_params** (*dict, optional*) – The parameters to override in the provided estimator/pipeline. It defaults to None.
- **field_delimiter** (*str, optional*) – The separation delimiter (comma for CSV, tab for TSV, etc.) used in the input train data set file. It defaults to ‘,’.
- **scorers** (*{list, dict}, optional*) – Multiple metrics to evaluate the predictions on the hold-out data. Either give a list of (unique) strings or a dict with names as keys and callables as values. The callables should be scorers built using scikit-learn `make_scorer`. Note that when using custom scorers, each scorer should return a single value. It defaults to [‘roc_auc’, ‘average_precision’, ‘f1’, ‘precision’, ‘recall’, ‘accuracy’].
- **feature_columns** (*{str, list}, optional*) – Either ‘all’ to use from the input train data set file all the columns or a list of column names to select specific columns. It defaults to ‘all’.
- **id_column** (*str, optional*) – The name of the column in the input train data set file containing the sample IDs. It defaults to ‘id’.
- **label_column** (*str, optional*) – The name of the column in the input train data set file containing the ground truth labels. It defaults to ‘label’.
- **objective_score** (*str, optional*) – The scorer that would be used to find the best hyperparameters for refitting the best estimator/pipeline at the end. It defaults to ‘f1’.
- **random_seed** (*int, optional*) – The random seed used in the random number generator. It can be used to reproduce the output. It defaults to 42.
- **verbose** (*int, optional*) – Controls the verbosity of output. The higher, the more messages. It defaults to 3.
- **n_jobs** (*int, optional*) – Number of jobs to run in parallel. -1 means using all processors. It defaults to 1.
- **return_train_score** (*bool, optional*) – If False, training scores will not be computed and returned. Computing training scores is used to get insights on how different parameter settings impact the overfitting/underfitting trade-off. It defaults to True.

grid_search()

Optional method.

Use the grid search method when searching the best hyperparameters.

Returns The object instance itself.

Return type *HyperParametersSearchCrossValidationTask*

random_search(*n_iters=200*)
Optional method.

Use the random search method when searching the best hyperparameters.

Parameters **n_iters** (*int, optional*) – Number of hyperparameter settings that are sampled. n_iters trades off runtime vs quality of the solution. It defaults to 200.

Returns The object instance itself.

Return type *HyperParametersSearchCrossValidationTask*

run(*output_directory*)
Run the task.

The search results (search_results) are stored also in a *search_results.html* file as a static HTML table under the output directory path.

Parameters **output_directory** (*str*) – The output directory path under which task-related generated files are stored.

Returns The task's result. Specifically, **1)** best_estimator: The estimator/pipeline that was chosen by the search, i.e. estimator/pipeline which gave best score on the hold-out data. **2)** best_params: The hyperparameters setting that gave the best results on the hold-out data. **3)** best_score: Mean cross-validated score of the best_estimator. **4)** search_results: Metrics measured for each of the hyperparameters setting in the search. **5)** best_index: The index (of the search_results) which corresponds to the best candidate hyperparameters setting.

Return type dict

custom_folds(*folds_data*, *fold_column='fold'*)
Optional method.

Use cross-validation with user-defined custom folds.

Parameters

- **folds_data** (*{str or pandas DataFrame}*) – The input folds data. It can be either a URL, a disk file path or a pandas DataFrame. The folds data contain the user-defined folds for the samples. If a URL or a disk file path is provided the data must be formatted with the same separation delimiter (comma for CSV, tab for TSV, etc.) as the one used in the input data set files provided to the task. The data must contain two columns and the first row must be the header. The first column is for the sample IDs and needs to be the same as the one used in the input data set files provided to the task. The second column is for the fold IDs (e.g., 1 through 5, A through D, etc.).
- **fold_column** (*str, optional*) – The column name for the fold IDs. It defaults to 'fold'.

Returns The object instance itself.

Return type *BaseCrossValidationTask*

get_configuration()
Get the task's parameters.

Returns The task's parameters as a dictionary of key-value pairs, where the key is the parameter name and the value is the parameter value.

Return type dict

get_type()
Get the task's type name.

Returns The task's type name.

Return type str

stratified_folds(*total_folds=3*, *shuffle=False*)
Optional method.

Use stratified k-fold cross-validation.

The folds are made by preserving the percentage of samples for each class.

Parameters

- **total_folds** (*int, optional*) – Number of folds. Must be at least 2. It defaults to 3.
- **shuffle** (*bool, optional*) – Whether to shuffle each class's samples before splitting into batches. Note that the samples within each split will not be shuffled. It defaults to False.

Returns The object instance itself.

Return type *BaseCrossValidationTask*

skrobot.tasks.deep_feature_synthesis_task module

class skrobot.tasks.deep_feature_synthesis_task.**DeepFeatureSynthesisTask**(*entities=None*, *relationships=None*, *entityset=None*, *target_entity=None*, *cutoff_time=None*, *instance_ids=None*, *agg_primitives=None*, *trans_primitives=None*, *groupby_trans_primitives*
allowed_paths=None, *max_depth=2*, *ignore_entities=None*, *ignore_variables=None*, *primitive_options=None*, *seed_features=None*, *drop_contains=None*, *drop_exact=None*, *where_primitives=None*, *max_features=-*

1, *save_progress=None*, *training_window=None*, *approximate=None*, *chunk_size=None*, *n_jobs=1*, *dask_kwargs=None*, *verbose=False*, *return_variable_types=Non*
progress_callback=None, *include_cutoff_time=True*, *export_feature_graphs=Fals*
export_feature_information=
label_column='label')

The `DeepFeatureSynthesisTask` class is a wrapper for Featuretools. It can be used to automate feature engineering and create features from temporal and relational datasets.

__init__(*entities=None*, *relationships=None*, *entityset=None*, *target_entity=None*, *cutoff_time=None*, *instance_ids=None*, *agg_primitives=None*, *trans_primitives=None*, *groupby_trans_primitives=None*, *allowed_paths=None*, *max_depth=2*, *ignore_entities=None*, *ignore_variables=None*, *primitive_options=None*, *seed_features=None*, *drop_contains=None*, *drop_exact=None*, *where_primitives=None*, *max_features=- 1*, *save_progress=None*, *training_window=None*, *approximate=None*, *chunk_size=None*, *n_jobs=1*, *dask_kwargs=None*, *verbose=False*, *return_variable_types=None*, *progress_callback=None*, *include_cutoff_time=True*, *export_feature_graphs=False*, *export_feature_information=False*, *label_column='label'*)

This is the constructor method and can be used to create a new object instance of `DeepFeatureSynthesisTask` class.

Most of the arguments are documented here: https://featuretools.alteryx.com/en/stable/generated/featuretools.dfs.html#featuretools.dfs

Parameters

- **export_feature_graphs** (*bool, optional*) – If this task will export feature computation graphs. It defaults to False.
- **export_feature_information** (*bool, optional*) – If this task will export feature information. The feature definitions can be used to recalculate features for a different data set. It defaults to False.
- **label_column** (*str, optional*) – The name of the column containing the ground truth labels. It defaults to 'label'.

run(*output_directory*)

Run the task.

The synthesized output data set is returned as a result and also stored in a *synthesized_dataset.csv* file under the output directory path.

The features information are stored in a *feature_information.html* file as a static HTML table under the output directory path.

The feature computation graphs are stored as PNG files under the output directory path.

Also, the feature definitions are stored in a *feature_definitions.txt* file under the output directory path.

Parameters **output_directory** (*str*) – The output directory path under which task-related generated files are stored.

Returns The task's result. Specifically, **1)** `synthesized_dataset`: The synthesized output data set as a pandas DataFrame. **2)** `feature_definitions`: The definitions of features in the synthesized output data set. The feature definitions can be used to recalculate features for a different data set.

Return type dict

get_configuration()

Get the task's parameters.

Returns The task's parameters as a dictionary of key-value pairs, where the key is the parameter name and the value is the parameter value.

Return type dict

get_type()

Get the task's type name.

Returns The task's type name.

Return type str

skrobot.tasks.dataset_calculation_task module

class skrobot.tasks.dataset_calculation_task.**DatasetCalculationTask**(*feature_definitions*, *entityset=None*, *cutoff_time=None*, *instance_ids=None*, *entities=None*, *relationships=None*, *training_window=None*, *approximate=None*, *save_progress=None*, *verbose=False*, *chunk_size=None*, *n_jobs=1*, *dask_kwargs=None*, *progress_callback=None*, *include_cutoff_time=True*)

Bases: *skrobot.tasks.base_task.BaseTask*

The *DatasetCalculationTask* class is a wrapper for Featuretools. It can be used to calculate a data set using some feature definitions and input data.

__init__(*feature_definitions*, *entityset=None*, *cutoff_time=None*, *instance_ids=None*, *entities=None*, *relationships=None*, *training_window=None*, *approximate=None*, *save_progress=None*, *verbose=False*, *chunk_size=None*, *n_jobs=1*, *dask_kwargs=None*, *progress_callback=None*, *include_cutoff_time=True*)

This is the constructor method and can be used to create a new object instance of *DatasetCalculationTask* class.

Most of the arguments are documented here: https://featuretools.alteryx.com/en/stable/generated/featuretools.calculate_feature_matrix.html#featuretools.calculate_feature_matrix

Parameters **feature_definitions** (*{str or list[FeatureBase]}*) – The feature definitions to be calculated. It can be either a disk file path or a list[FeatureBase] as exported by *DeepFeatureSynthesisTask* task.

run(*output_directory*)

Run the task.

The calculated data set is returned as a result.

Parameters **output_directory** (*str*) – The output directory path under which task-related generated files are stored.

Returns The task's result. Specifically, the calculated data set for the input data and feature definitions.

Return type pandas DataFrame

get_configuration()
Get the task's parameters.

Returns The task's parameters as a dictionary of key-value pairs, where the key is the parameter name and the value is the parameter value.

Return type dict

get_type()
Get the task's type name.

Returns The task's type name.

Return type str

skrobot.tasks.train_task module

class skrobot.tasks.train_task.**TrainTask**(*estimator*, *train_data_set*, *estimator_params=None*, *field_delimiter=','*, *feature_columns='all'*, *id_column='id'*, *label_column='label'*, *random_seed=42*)
Bases: *skrobot.tasks.base_task.BaseTask*

The *TrainTask* class can be used to fit a scikit-learn estimator/pipeline on train data.

__init__(*estimator*, *train_data_set*, *estimator_params=None*, *field_delimiter=','*, *feature_columns='all'*, *id_column='id'*, *label_column='label'*, *random_seed=42*)
This is the constructor method and can be used to create a new object instance of *TrainTask* class.

Parameters

- **estimator** (*scikit-learn {estimator, pipeline}*) – It can be either an estimator (e.g., LogisticRegression) or a pipeline ending with an estimator.
- **train_data_set** (*{str, pandas DataFrame}*) – The input train data set. It can be either a URL, a disk file path or a pandas DataFrame.
- **estimator_params** (*dict, optional*) – The parameters to override in the provided estimator/pipeline. It defaults to None.
- **field_delimiter** (*str, optional*) – The separation delimiter (comma for CSV, tab for TSV, etc.) used in the input train data set file. It defaults to ','.
- **feature_columns** (*{str, list}, optional*) – Either 'all' to use from the input train data set file all the columns or a list of column names to select specific columns. It defaults to 'all'.
- **id_column** (*str, optional*) – The name of the column in the input train data set file containing the sample IDs. It defaults to 'id'.
- **label_column** (*str, optional*) – The name of the column in the input train data set file containing the ground truth labels. It defaults to 'label'.
- **random_seed** (*int, optional*) – The random seed used in the random number generator. It can be used to reproduce the output. It defaults to 42.

run(*output_directory*)
Run the task.

The fitted estimator/pipeline is returned as a result and also stored in a *trained_model.pkl* pickle file under the output directory path.

Parameters **output_directory** (*str*) – The output directory path under which task-related generated files are stored.

Returns The task's result. Specifically, the fitted estimator/pipeline.

Return type dict

get_configuration()
Get the task's parameters.

Returns The task's parameters as a dictionary of key-value pairs, where the key is the parameter name and the value is the parameter value.

Return type dict

get_type()
Get the task's type name.

Returns The task's type name.

Return type str

skrobot.tasks.prediction_task module

class skrobot.tasks.prediction_task.**PredictionTask**(*estimator*, *data_set*, *field_delimiter=','*, *feature_columns='all'*, *id_column='id'*, *prediction_column='prediction'*, *threshold=0.5*)
Bases: *skrobot.tasks.base_task.BaseTask*

The *PredictionTask* class can be used to predict new data using a scikit-learn estimator/pipeline.

__init__(*estimator*, *data_set*, *field_delimiter=','*, *feature_columns='all'*, *id_column='id'*, *prediction_column='prediction'*, *threshold=0.5*)
This is the constructor method and can be used to create a new object instance of *PredictionTask* class.

Parameters

- **estimator** (*scikit-learn {estimator, pipeline}*) – It can be either an estimator (e.g., LogisticRegression) or a pipeline ending with an estimator. The estimator needs to be able to predict probabilities through a `predict_proba` method.
- **data_set** (*{str, pandas DataFrame}*) – The input data set. It can be either a URL, a disk file path or a pandas DataFrame.
- **field_delimiter** (*str, optional*) – The separation delimiter (comma for CSV, tab for TSV, etc.) used in the input data set file. It defaults to ','.
- **feature_columns** (*{str, list}, optional*) – Either 'all' to use from the input data set file all the columns or a list of column names to select specific columns. It defaults to 'all'.

- **id_column** (*str, optional*) – The name of the column in the input data set file containing the sample IDs. It defaults to 'id'.
- **prediction_column** (*str, optional*) – The name of the column for the predicted binary class labels. It defaults to 'prediction'.
- **threshold** (*float, optional*) – The threshold to use for converting the predicted probability into a binary class label. It defaults to 0.5.

run(*output_directory*)
Run the task.

The predictions are returned as a result and also stored in a *predictions.csv* CSV file under the output directory path.

Parameters **output_directory** (*str*) – The output directory path under which task-related generated files are stored.

Returns The task's result. Specifically, the predictions for the input data set, containing the sample IDs, the predicted binary class labels, and the predicted probabilities for the positive class.

Return type pandas DataFrame

get_configuration()
Get the task's parameters.

Returns The task's parameters as a dictionary of key-value pairs, where the key is the parameter name and the value is the parameter value.

Return type dict

get_type()
Get the task's type name.

Returns The task's type name.

Return type str

CHAPTER

TWO

WHAT IS IT ABOUT?

skrobot is a Python module for designing, running and tracking Machine Learning experiments / tasks. It is built on top of scikit-learn framework.

CHAPTER

THREE

WHY DOES IT EXISTS?

It can help Data Scientists and Machine Learning Engineers:

- to keep track of modelling experiments / tasks
- to automate the repetitive (and boring) stuff when designing modelling pipelines
- to spend more time on the things that truly matter when solving a problem

CHAPTER

FOUR

HOW DO I INSTALL IT?

4.1 PyPI

```
pip install skrobot
```

4.2 Graphviz

If you want to export feature computation graphs using the argument `export_feature_graphs` in *DeepFeatureSynthesisTask* class, you need to install Graphviz.

Conda users:

```
conda install python-graphviz
```

GNU/Linux:

```
sudo apt-get install graphviz
pip install graphviz
```

Mac OS:

```
brew install graphviz
pip install graphviz
```

Windows:

```
conda install python-graphviz
```

4.3 Development Version

The skrobot version on PyPI may always be one step behind; you can install the latest development version from the GitHub repository by executing

```
pip install git+git://github.com/medoidai/skrobot.git
```

Or, you can clone the GitHub repository and install skrobot from your local drive via

```
pip install .
```

CHAPTER

FIVE

WHICH ARE THE COMPONENTS?

NOTE : Currently, skrobot can be used only for binary classification problems.

5.1 For the module's users

Component	What is this?
Train Task	This task can be used to fit a scikit-learn estimator on some data.
Prediction Task	This task can be used to predict new data using a scikit-learn estimator.
Evaluation Cross Validation Task	This task can be used to evaluate a scikit-learn estimator on some data.
Deep Feature Synthesis Task	This task can be used to automate feature engineering and create features from temporal and relational datasets.
Dataset Calculation Task	This task can be used to calculate a data set using some feature definitions and input data.
Feature Selection Cross Validation Task	This task can be used to perform feature selection with Recursive Feature Elimination using a scikit-learn estimator on some data.
Hyperparameters Search Cross Validation Task	This task can be used to search the best hyperparameters of a scikit-learn estimator on some data.
Email Notifier	This notifier can be used to send email notifications.
Experiment	This is used to build, track and run an experiment. It can run tasks in the context of an experiment.
Task Runner	This is a simplified version (in functionality) of the Experiment component. It leaves out all the "experiment" stuff and is focused mostly in the execution and tracking of tasks.

5.2 For the module's developers

Component	What is this?
Base Task	All tasks inherit from this component. A task is a configurable and reproducible piece of code built on top of scikit-learn that can be used in machine learning pipelines.
Base Cross Validation Task	All tasks that use cross validation functionality inherit from this component.
Base Notifier	All notifiers inherit from this component. A notifier can be used to send success / failure notifications for tasks execution.

CHAPTER

SIX

HOW DO I USE IT?

The following examples use many of skrobot's components to built a machine learning modelling pipeline. Please try them and we would love to have your feedback! Furthermore, many examples can be found in the project's repository.

6.1 Example on Titanic Dataset

The following example has generated the following results.

```
import os

import pandas as pd

from sklearn.compose import ColumnTransformer
from sklearn.pipeline import Pipeline
from sklearn.impute import SimpleImputer
from sklearn.preprocessing import StandardScaler, OneHotEncoder
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import train_test_split

from skrobot.core import Experiment
from skrobot.tasks import TrainTask
from skrobot.tasks import PredictionTask
from skrobot.tasks import FeatureSelectionCrossValidationTask
from skrobot.tasks import EvaluationCrossValidationTask
from skrobot.tasks import HyperParametersSearchCrossValidationTask
from skrobot.tasks import DeepFeatureSynthesisTask
from skrobot.tasks import DatasetCalculationTask
from skrobot.feature_selection import ColumnSelector
from skrobot.notification import EmailNotifier

######### Initialization Code

id_column = 'PassengerId'

label_column = 'Survived'

numerical_columns = [ 'Age', 'Fare', 'SibSp', 'Parch' ]

categorical_columns = [ 'Embarked', 'Sex', 'Pclass' ]

columns_subset = numerical_columns + categorical_columns

raw_data_set = pd.read_csv('https://bit.ly/titanic-data-set', usecols=[id_column,␣
↪label_column, *columns_subset], dtype={ c: 'category' for c in categorical_columns}␣
↪)
```

(continues on next page)

(continued from previous page)

```
new_raw_data_set = pd.read_csv('https://bit.ly/titanic-data-new', usecols=[id_column,␣
↪*columns_subset], dtype={ c: 'category' for c in categorical_columns })

random_seed = 42

test_ratio = 0.2

numeric_transformer = Pipeline(steps=[
    ('imputer', SimpleImputer()),
    ('scaler', StandardScaler())])

categorical_transformer = Pipeline(steps=[
    ('imputer', SimpleImputer(strategy='most_frequent')),
    ('encoder', OneHotEncoder(handle_unknown='ignore'))])

classifier = LogisticRegression(solver='liblinear', random_state=random_seed)

search_params = {
    "classifier__C" : [ 1.e-01, 1.e+00, 1.e+01 ],
    "classifier__penalty" : [ "l1", "l2" ],
    "preprocessor__numerical_transformer__imputer__strategy" : [ "mean", "median" ]
}

######### skrobot Code

# Create a Notifier
notifier = EmailNotifier(email_subject="skrobot notification",
                         sender_account=os.environ['EMAIL_SENDER_ACCOUNT'],
                         sender_password=os.environ['EMAIL_SENDER_PASSWORD'],
                         smtp_server=os.environ['EMAIL_SMTP_SERVER'],
                         smtp_port=os.environ['EMAIL_SMTP_PORT'],
                         recipients=os.environ['EMAIL_RECIPIENTS'])

# Build an Experiment
experiment = Experiment('experiments-output').set_source_code_file_path(__file__).set_
↪experimenter('echatzikyriakidis').set_notifier(notifier).build()

# Run Deep Feature Synthesis Task
feature_synthesis_results = experiment.run(DeepFeatureSynthesisTask (entities={
↪"passengers" : (raw_data_set, id_column) },
                                                                     target_entity=
↪"passengers",
                                                                     trans_primitives␣
↪= ['add_numeric', 'multiply_numeric'],
                                                                     export_feature_
↪information=True,
                                                                     export_feature_
↪graphs=True,
                                                                     label_
↪column=label_column))

data_set = feature_synthesis_results['synthesized_dataset']

feature_defs = feature_synthesis_results['feature_definitions']

train_data_set, test_data_set = train_test_split(data_set, test_size=test_ratio,␣
↪stratify=data_set[label_column], random_state=random_seed)
```

(continues on next page)

(continued from previous page)

```
numerical_features = [ o.get_name() for o in feature_defs if any([ x in o.get_name()␣
↪for x in numerical_columns])]

categorical_features = [ o.get_name() for o in feature_defs if any([ x in o.get_
↪name() for x in categorical_columns])]

preprocessor = ColumnTransformer(transformers=[
    ('numerical_transformer', numeric_transformer, numerical_features),
    ('categorical_transformer', categorical_transformer, categorical_features)])

# Run Feature Selection Task
features_columns = experiment.run(FeatureSelectionCrossValidationTask␣
↪(estimator=classifier,
                                                                    train_data_
↪set=train_data_set,
                                                                   ␣
↪preprocessor=preprocessor,
                                                                    id_column=id_
↪column,
                                                                    label_
↪column=label_column,
                                                                    random_
↪seed=random_seed).stratified_folds(total_folds=5, shuffle=True))

pipe = Pipeline(steps=[('preprocessor', preprocessor),
                       ('selector', ColumnSelector(cols=features_columns)),
                       ('classifier', classifier)])

# Run Hyperparameters Search Task
hyperparameters_search_results = experiment.
↪run(HyperParametersSearchCrossValidationTask (estimator=pipe,
↪     search_params=search_params,
↪     train_data_set=train_data_set,
↪     id_column=id_column,
↪     label_column=label_column,
↪     random_seed=random_seed).random_search(n_iters=100).stratified_folds(total_
↪folds=5, shuffle=True))

# Run Evaluation Task
evaluation_results = experiment.run(EvaluationCrossValidationTask(estimator=pipe,
                                                                  estimator_
↪params=hyperparameters_search_results['best_params'],
                                                                  train_data_
↪set=train_data_set,
                                                                  test_data_set=test_
↪data_set,
                                                                  id_column=id_column,
                                                                  label_column=label_
↪column,
                                                                  random_seed=random_
↪seed,
```

(continues on next page)

(continued from previous page)

```
                                                                    export_
↪classification_reports=True,
                                                                    export_confusion_
↪matrixes=True,
                                                                    export_pr_
↪curves=True,
                                                                    export_roc_
↪curves=True,
                                                                    export_false_
↪positives_reports=True,
                                                                    export_false_
↪negatives_reports=True,
                                                                    export_also_for_
↪train_folds=True).stratified_folds(total_folds=5, shuffle=True))

# Run Train Task
train_results = experiment.run(TrainTask(estimator=pipe,
                                         estimator_params=hyperparameters_search_
↪results['best_params'],
                                         train_data_set=train_data_set,
                                         id_column=id_column,
                                         label_column=label_column,
                                         random_seed=random_seed))

# Run Dataset Calculation Task
new_data_set = experiment.run(DatasetCalculationTask(feature_defs, entities={
↪"passengers" : (new_raw_data_set, id_column) }))

# Run Prediction Task
predictions = experiment.run(PredictionTask(estimator=train_results['estimator'],
                                            data_set=new_data_set,
                                            id_column=id_column,
                                            prediction_column=label_column,
                                            threshold=evaluation_results['threshold
↪']))

# Print in-memory results
print(feature_synthesis_results['synthesized_dataset'])
print(feature_synthesis_results['feature_definitions'])

print(features_columns)

print(hyperparameters_search_results['best_params'])
print(hyperparameters_search_results['best_index'])
print(hyperparameters_search_results['best_estimator'])
print(hyperparameters_search_results['best_score'])
print(hyperparameters_search_results['search_results'])

print(evaluation_results['threshold'])
print(evaluation_results['cv_threshold_metrics'])
print(evaluation_results['cv_splits_threshold_metrics'])
print(evaluation_results['cv_splits_threshold_metrics_summary'])
print(evaluation_results['test_threshold_metrics'])

print(train_results['estimator'])

print(predictions)
```

6.2 Example on SMS Spam Collection Dataset

The following example has generated the following results.

```
from sklearn.pipeline import Pipeline
from sklearn.feature_extraction.text import CountVectorizer, TfidfTransformer
from sklearn.feature_selection import SelectPercentile, chi2
from sklearn.linear_model import SGDClassifier

from skrobot.core import Experiment
from skrobot.tasks import TrainTask
from skrobot.tasks import PredictionTask
from skrobot.tasks import EvaluationCrossValidationTask
from skrobot.tasks import HyperParametersSearchCrossValidationTask
from skrobot.feature_selection import ColumnSelector

######### Initialization Code

train_data_set = 'https://bit.ly/sms-spam-ham-data-train'

test_data_set = 'https://bit.ly/sms-spam-ham-data-test'

new_data_set = 'https://bit.ly/sms-spam-ham-data-new'

field_delimiter = '\t'

random_seed = 42

pipe = Pipeline(steps=[
    ('column_selection', ColumnSelector(cols=['message'], drop_axis=True)),
    ('vectorizer', CountVectorizer()),
    ('tfidf', TfidfTransformer()),
    ('feature_selection', SelectPercentile(chi2)),
    ('classifier', SGDClassifier(loss='log'))])

search_params = {
    'classifier__max_iter': [ 20, 50, 80 ],
    'classifier__alpha': [ 0.00001, 0.000001 ],
    'classifier__penalty': [ 'l2', 'elasticnet' ],
    "vectorizer__stop_words" : [ "english", None ],
    "vectorizer__ngram_range" : [ (1, 1), (1, 2) ],
    "vectorizer__max_df": [ 0.5, 0.75, 1.0 ],
    "tfidf__use_idf" : [ True, False ],
    "tfidf__norm" : [ 'l1', 'l2' ],
    "feature_selection__percentile" : [ 70, 60, 50 ]
}

######### skrobot Code

# Build an Experiment
experiment = Experiment('experiments-output').set_source_code_file_path(__file__).set_
↪experimenter('echatzikyriakidis').build()

# Run Hyperparameters Search Task
hyperparameters_search_results = experiment.
↪run(HyperParametersSearchCrossValidationTask (estimator=pipe,

↪     search_params=search_params,
```

(continues on next page)

(continued from previous page)

```
↪    train_data_set=train_data_set,
↪    field_delimiter=field_delimiter,
↪    random_seed=random_seed).random_search().stratified_folds(total_folds=5,␣
↪shuffle=True))

# Run Evaluation Task
evaluation_results = experiment.run(EvaluationCrossValidationTask(estimator=pipe,
                                                                  estimator_
↪params=hyperparameters_search_results['best_params'],
                                                                  train_data_
↪set=train_data_set,
                                                                  test_data_set=test_
↪data_set,
                                                                  field_
↪delimiter=field_delimiter,
                                                                  random_seed=random_
↪seed,
                                                                  export_
↪classification_reports=True,
                                                                  export_confusion_
↪matrixes=True,
                                                                  export_pr_
↪curves=True,
                                                                  export_roc_
↪curves=True,
                                                                  export_false_
↪positives_reports=True,
                                                                  export_false_
↪negatives_reports=True,
                                                                  export_also_for_
↪train_folds=True).stratified_folds(total_folds=5, shuffle=True))

# Run Train Task
train_results = experiment.run(TrainTask(estimator=pipe,
                                         estimator_params=hyperparameters_search_
↪results['best_params'],
                                         train_data_set=train_data_set,
                                         field_delimiter=field_delimiter,
                                         random_seed=random_seed))

# Run Prediction Task
predictions = experiment.run(PredictionTask(estimator=train_results['estimator'],
                                            data_set=new_data_set,
                                            field_delimiter=field_delimiter,
                                            threshold=evaluation_results['threshold
↪']))

# Print in-memory results
print(hyperparameters_search_results['best_params'])
print(hyperparameters_search_results['best_index'])
print(hyperparameters_search_results['best_estimator'])
print(hyperparameters_search_results['best_score'])
print(hyperparameters_search_results['search_results'])
```

(continues on next page)

(continued from previous page)

```
print(evaluation_results['threshold'])
print(evaluation_results['cv_threshold_metrics'])
print(evaluation_results['cv_splits_threshold_metrics'])
print(evaluation_results['cv_splits_threshold_metrics_summary'])
print(evaluation_results['test_threshold_metrics'])

print(train_results['estimator'])

print(predictions)
```

CHAPTER

SEVEN

SAMPLE OF GENERATED RESULTS?

7.1 Classification Reports

7.2 Confusion Matrixes

7.3 False Negatives

7.4 False Positives

7.5 PR Curves

7.6 ROC Curves

7.7 Performance Metrics

On train / validation CV folds:

On hold-out test set:

7.8 Hyperparameters Search Results

7.9 Task Parameters Logging

7.10 Experiment Logging

7.11 Features Selected

The selected column indexes from the transformed features (this is generated when a preprocessor is used):

The selected column names from the original features (this is generated when no preprocessor is used):

7.12 Experiment Source Code

7.13 Predictions

7.14 Synthesized Features Information

7.15 Synthesized Features Definition

7.16 Synthesized Output Dataset

7.17 Synthesized Features Computation Graphs

CHAPTER

EIGHT

THE PEOPLE BEHIND IT?

Development:

- Efstathios Chatzikyriakidis

Support, testing and features recommendation:

- Lefteris Kouloubris
- Antonis Markou
- Christina Chrysouli
- Michalis Chaviaras

And last but not least, all the open-source contributors whose work went into RELEASES.

CHAPTER

NINE

CAN I CONTRIBUTE?

Of course, the project is Free Software and you can contribute to it!

CHAPTER

TEN

WHAT LICENSE DO YOU USE?

See our LICENSE for more details.

PYTHON MODULE INDEX

S

INDEX

Symbols

B

C

D

E

F

G

H

M

N

P

R

S

T