SciKit-Learn Laboratory Documentation

Release 5.0.1

Educational Testing Service

Mar 08, 2024

CONTENTS

SKLL (pronounced “skull”) provides a number of utilities to make it simpler to run common scikit-learn experiments with pre-generated features.

There are two primary means of using SKLL: *the run_experiment script* and *the Python API*.

CHAPTER ONE

DOCUMENTATION

1.1 Installation

SKLL can be installed via `pip`:

```
pip install skll
```

or via conda:

```
conda install -c conda-forge -c ets skll
```

It can also be downloaded directly from GitHub.

1.2 License

SKLL is distributed under the 3-clause BSD License.

1.3 Tutorial

Before doing anything below, you'll want to *install SKLL*.

1.3.1 Workflow

In general, there are four steps to using SKLL:

1. Get some data in a *SKLL-compatible format*.
2. Create a small *configuration file* describing the machine learning experiment you would like to run.

3. Run that configuration file with *run_experiment*.
4. Examine the results of the experiment.

1.3.2 Titanic Example

Let's see how we can apply the basic workflow above to a simple example using the Titanic: Machine Learning from Disaster data from Kaggle.

Create virtual environment with SKLL

Before we proceed further, we need to install SKLL. The easiest way to do this is in a virtual environment. For this tutorial, we will use conda for creating our virtual environment as follows:

```
conda create -n skllenv -c conda-forge -c ets python=3.11 skll
```

This will create a new virtual environment named `skllenv` with the latest release of SKLL which you can then activate by running `conda activate skllenv`. Make sure to create and activate this environment before proceeding further. Once you are done with the tutorial, you may deactivate the virtual environment by running `conda deactivate`.

Get your data into the correct format

The first step is to get the Titanic data. We have already downloaded the data files from Kaggle and included them in the SKLL repository. Next, we need to get the files and process them to get them in the right shape.

The provided script, `make_titanic_example_data.py`, will split the train and test data files from Kaggle up into groups of related features and store them in `dev`, `test`, `train`, and `train+dev` subdirectories. The development set that gets created by the script is 20% of the data that was in the original training set, and `train` contains the other 80%.

Create a configuration file for the experiment

For this tutorial, we will refer to an "experiment" as having a single data set split into training and testing portions. As part of each experiment, we can train and test several models, either simultaneously or sequentially, depending whether we're using GridMap or not. This will be described in more detail later on, when we are ready to run our experiment.

You can consult the *full list of learners currently available* in SKLL to get an idea for the things you can do. As part of this tutorial, we will use the following classifiers:

- Decision Tree

- Multinomial Naïve Bayes
- Random Forest
- Support Vector Machine

```
[General]
experiment_name = Titanic_Evaluate_Tuned
task = evaluate

[Input]
# this could also be an absolute path instead (and must be if you're not
# running things in local mode)
train_directory = train
test_directory = dev
featuresets = [["family.csv", "misc.csv", "socioeconomic.csv", "vitals.csv
↪"]]
learners = ["RandomForestClassifier", "DecisionTreeClassifier", "SVC",
↪"MultinomialNB"]
label_col = Survived
id_col = PassengerId

[Tuning]
grid_search = true
grid_search_folds = 3
objectives = ['accuracy']

[Output]
# again, these can be absolute paths
metrics = ['roc_auc']
probability = true
logs = output
results = output
predictions = output
models = output
```

Let's take a look at the options specified in `titanic/evaluate_tuned.cfg`. Here, we are only going to train a model and evaluate its performance on the development set, because in the *General* section, *task* is set to *evaluate*. We will explore the other options for *task* later.

In the *Input* section, we have specified relative paths to the training and testing directories via the *train_directory* and *test_directory* settings respectively. *featuresets* indicates the name of both the training and testing files. *learners* must always be specified in between `[ ]` brackets, even if you only want to use one learner. This is similar to the *featuresets* option, which requires two sets of brackets, since multiple sets of different-yet-related features can be provided. We will keep our examples simple, however, and only use one set of features per experiment. The *label_col* and *id_col* settings specify the columns in the CSV files that specify the class labels and instances IDs for each example.

The *Tuning* section defines how we want our model to be tuned. Setting *grid_search* to `True` here employs scikit-learn's GridSearchCV class, which is an implementation of the standard, brute-force approach to hyperparameter optimization.

objectives refers to the desired objective functions; here, `accuracy` will optimize for overall accuracy. You can see a list of all the available objective functions *here*.

In the *Output* section, we first define the additional evaluation metrics we want to compute in addition to the tuning objective via the *metrics* option. The other options are directories where you'd like all of the relevant output from your experiment to go. *results* refers to the results of the experiment in both human-readable and JSON forms. *logs* specifies where to put log files containing any status, warning, or error messages generated during model training and evaluation. *predictions* refers to where to store the individual predictions generated for the test set. *models* is for specifying a directory to serialize the trained models.

Running your configuration file through run_experiment

Getting your experiment running is the simplest part of using SKLL, you just need to type the following into a terminal:

```
$ run_experiment titanic/evaluate_tuned.cfg
```

Make sure you have the `skllenv` environment activated before you run this command which should produce output like:

```
2020-03-10 14:25:23,596 - Titanic_Evaluate_Tuned_family.csv+misc.
↪csv+socioeconomic.csv+vitals.csv_RandomForestClassifier - INFO - Task:␣
↪evaluate
2020-03-10 14:25:23,596 - Titanic_Evaluate_Tuned_family.csv+misc.
↪csv+socioeconomic.csv+vitals.csv_RandomForestClassifier - INFO -␣
↪Training on train, Test on dev, feature set ['family.csv', 'misc.csv',
↪'socioeconomic.csv', 'vitals.csv'] ...
Loading /Users/nmadnani/work/skll/examples/titanic/train/family.csv...    ␣
↪       done
Loading /Users/nmadnani/work/skll/examples/titanic/train/misc.csv...      ␣
↪     done
Loading /Users/nmadnani/work/skll/examples/titanic/train/socioeconomic.
↪csv...           done
Loading /Users/nmadnani/work/skll/examples/titanic/train/vitals.csv...    ␣
↪       done
Loading /Users/nmadnani/work/skll/examples/titanic/dev/family.csv...      ␣
↪     done
```

(continues on next page)

(continued from previous page)

```
Loading /Users/nmadnani/work/skll/examples/titanic/dev/misc.csv...      ␣
↪   done
Loading /Users/nmadnani/work/skll/examples/titanic/dev/socioeconomic.csv..
↪.         done
Loading /Users/nmadnani/work/skll/examples/titanic/dev/vitals.csv...    ␣
↪     done
2020-03-10 14:25:23,662 - Titanic_Evaluate_Tuned_family.csv+misc.
↪csv+socioeconomic.csv+vitals.csv_RandomForestClassifier - INFO -␣
↪Featurizing and training new RandomForestClassifier model
2020-03-10 14:25:23,663 - Titanic_Evaluate_Tuned_family.csv+misc.
↪csv+socioeconomic.csv+vitals.csv_RandomForestClassifier - WARNING -␣
↪Training data will be shuffled to randomize grid search folds. ␣
↪Shuffling may yield different results compared to scikit-learn.
2020-03-10 14:25:28,129 - Titanic_Evaluate_Tuned_family.csv+misc.
↪csv+socioeconomic.csv+vitals.csv_RandomForestClassifier - INFO - Best␣
↪accuracy grid search score: 0.798
2020-03-10 14:25:28,130 - Titanic_Evaluate_Tuned_family.csv+misc.
↪csv+socioeconomic.csv+vitals.csv_RandomForestClassifier - INFO -␣
↪Hyperparameters: bootstrap: True, ccp_alpha: 0.0, class_weight: None,␣
↪criterion: gini, max_depth: 5, max_features: auto, max_leaf_nodes: None,
↪ max_samples: None, min_impurity_decrease: 0.0, min_impurity_split:␣
↪None, min_samples_leaf: 1, min_samples_split: 2, min_weight_fraction_
↪leaf: 0.0, n_estimators: 500, n_jobs: None, oob_score: False, random_
↪state: 123456789, verbose: 0, warm_start: False
2020-03-10 14:25:28,130 - Titanic_Evaluate_Tuned_family.csv+misc.
↪csv+socioeconomic.csv+vitals.csv_RandomForestClassifier - INFO -␣
↪Evaluating predictions
2020-03-10 14:25:28,172 - Titanic_Evaluate_Tuned_family.csv+misc.
↪csv+socioeconomic.csv+vitals.csv_RandomForestClassifier - INFO - using␣
↪probabilities for the positive class to compute "roc_auc" for␣
↪evaluation.
2020-03-10 14:25:28,178 - Titanic_Evaluate_Tuned_family.csv+misc.
↪csv+socioeconomic.csv+vitals.csv_DecisionTreeClassifier - INFO - Task:␣
↪evaluate
2020-03-10 14:25:28,178 - Titanic_Evaluate_Tuned_family.csv+misc.
↪csv+socioeconomic.csv+vitals.csv_DecisionTreeClassifier - INFO -␣
↪Training on train, Test on dev, feature set ['family.csv', 'misc.csv',
↪'socioeconomic.csv', 'vitals.csv'] ...
Loading /Users/nmadnani/work/skll/examples/titanic/train/family.csv...  ␣
↪       done
Loading /Users/nmadnani/work/skll/examples/titanic/train/misc.csv...    ␣
↪     done
Loading /Users/nmadnani/work/skll/examples/titanic/train/socioeconomic.
```

(continues on next page)

(continued from previous page)

```
↪csv...           done
Loading /Users/nmadnani/work/skll/examples/titanic/train/vitals.csv...    ␣
↪       done
Loading /Users/nmadnani/work/skll/examples/titanic/dev/family.csv...      ␣
↪     done
Loading /Users/nmadnani/work/skll/examples/titanic/dev/misc.csv...        ␣
↪   done
Loading /Users/nmadnani/work/skll/examples/titanic/dev/socioeconomic.csv..
↪.           done
Loading /Users/nmadnani/work/skll/examples/titanic/dev/vitals.csv...      ␣
↪     done
2020-03-10 14:25:28,226 - Titanic_Evaluate_Tuned_family.csv+misc.
↪csv+socioeconomic.csv+vitals.csv_DecisionTreeClassifier - INFO -␣
↪Featurizing and training new DecisionTreeClassifier model
2020-03-10 14:25:28,226 - Titanic_Evaluate_Tuned_family.csv+misc.
↪csv+socioeconomic.csv+vitals.csv_DecisionTreeClassifier - WARNING -␣
↪Training data will be shuffled to randomize grid search folds. ␣
↪Shuffling may yield different results compared to scikit-learn.
2020-03-10 14:25:28,269 - Titanic_Evaluate_Tuned_family.csv+misc.
↪csv+socioeconomic.csv+vitals.csv_DecisionTreeClassifier - INFO - Best␣
↪accuracy grid search score: 0.754
2020-03-10 14:25:28,269 - Titanic_Evaluate_Tuned_family.csv+misc.
↪csv+socioeconomic.csv+vitals.csv_DecisionTreeClassifier - INFO -␣
↪Hyperparameters: ccp_alpha: 0.0, class_weight: None, criterion: gini,␣
↪max_depth: None, max_features: None, max_leaf_nodes: None, min_impurity_
↪decrease: 0.0, min_impurity_split: None, min_samples_leaf: 1, min_
↪samples_split: 2, min_weight_fraction_leaf: 0.0, presort: deprecated,␣
↪random_state: 123456789, splitter: best
2020-03-10 14:25:28,269 - Titanic_Evaluate_Tuned_family.csv+misc.
↪csv+socioeconomic.csv+vitals.csv_DecisionTreeClassifier - INFO -␣
↪Evaluating predictions
2020-03-10 14:25:28,272 - Titanic_Evaluate_Tuned_family.csv+misc.
↪csv+socioeconomic.csv+vitals.csv_DecisionTreeClassifier - INFO - using␣
↪probabilities for the positive class to compute "roc_auc" for␣
↪evaluation.
```

We could squelch the warnings about shuffling by setting *shuffle* to `True` in the *Input* section.

The reason we see the loading messages repeated is that we are running the different learners sequentially, whereas SKLL is designed to take advantage of a cluster to execute everything in parallel via GridMap.

Examine the results

As a result of running our experiment, there will be a whole host of files in our *results* directory. They can be broken down into three types of files:

1. `.results` files, which contain a human-readable summary of the experiment, complete with confusion matrix.
2. `.results.json` files, which contain all of the same information as the `.results` files, but in a format more well-suited to automated processing.
3. A summary `.tsv` file, which contains all of the information in all of the `.results.json` files with one line per file. This is very nice if you're trying many different learners and want to compare their performance. If you do additional experiments later (with a different config file), but would like one giant summary file, you can use the *summarize_results* command.

An example of a human-readable results file for our Titanic experiment is:

```
Experiment Name: Titanic_Evaluate_Tuned
SKLL Version: 2.1
Training Set: train
Training Set Size: 569
Test Set: dev
Test Set Size: 143
Shuffle: False
Feature Set: ["family.csv", "misc.csv", "socioeconomic.csv", "vitals.csv"]
Learner: RandomForestClassifier
Task: evaluate
Feature Scaling: none
Grid Search: True
Grid Search Folds: 3
Grid Objective Function: accuracy
Additional Evaluation Metrics: ['roc_auc']
Scikit-learn Version: 0.22.2.post1
Start Timestamp: 10 Mar 2020 14:25:23.595787
End Timestamp: 10 Mar 2020 14:25:28.175375
Total Time: 0:00:04.579588


Fold:
Model Parameters: {"bootstrap": true, "ccp_alpha": 0.0, "class_weight":␣
↪null, "criterion": "gini", "max_depth": 5, "max_features": "auto", "max_
↪leaf_nodes": null, "max_samples": null, "min_impurity_decrease": 0.0,
↪"min_impurity_split": null, "min_samples_leaf": 1, "min_samples_split":␣
↪2, "min_weight_fraction_leaf": 0.0, "n_estimators": 500, "n_jobs": null,
↪ "oob_score": false, "random_state": 123456789, "verbose": 0, "warm_
```

(continues on next page)

(continued from previous page)

```
↪start": false}
Grid Objective Score (Train) = 0.797874315418175
+----+------+------+-------------+----------+-------------+
|    |    0 |    1 |   Precision |   Recall |   F-measure |
+====+======+======+=============+==========+=============+
|  0 | [79] |    8 |       0.849 |    0.908 |       0.878 |
+----+------+------+-------------+----------+-------------+
|  1 |   14 | [42] |       0.840 |    0.750 |       0.792 |
+----+------+------+-------------+----------+-------------+
(row = reference; column = predicted)
Accuracy = 0.8461538461538461
Objective Function Score (Test) = 0.8461538461538461

Additional Evaluation Metrics (Test):
 roc_auc = 0.9224137931034483
```

1.3.3 IRIS Example on Binder

If you prefer using an interactive Jupyter notebook to learn about SKLL, you can do so by clicking the launch button below.

1.4 Running Experiments

1.4.1 General Workflow

To run your own SKLL experiments via the command line, the following general workflow is recommended.

Get your data into the correct format

SKLL can work with several common data formats, all of which are described *here*.

If you need to convert between any of the supported formats, because, for example, you would like to create a single data file that will work both with SKLL and Weka (or some other external tool), the *skll_convert* script can help you out. It is as easy as:

```
$ skll_convert examples/titanic/train/family.csv examples/titanic/train/
↪family.arff
```

Create sparse feature files, if necessary

skll_convert can also create sparse data files in *.jsonlines*, *.libsvm*, or *.ndj* formats. This is very useful for saving disk space and memory when you have a large data set with mostly zero-valued features.

Set up training and testing directories/files

At a minimum, you will probably want to work with a training set and a testing set. If you have multiple feature files that you would like SKLL to join together for you automatically, you will need to create feature files with the exact same names and store them in training and testing directories. You can specifiy these directories in your config file using *train_directory* and *test_directory*. The list of files is specified using the *featuresets* setting.

If you're conducting a simpler experiment, where you have a single training file with all of your features and a similar single testing file, you should use the *train_file* and *test_file* settings in your config file.

Note: If you would like to split an existing file up into a training set and a testing set, you can employ the *filter_features* utility script to select instances you would like to include in each file.

Create an experiment configuration file

You saw a *basic configuration file* in the tutorial. For your own experiment, you will need to refer to the *Configuration file fields* section.

Run configuration file through run_experiment

There are a few meta-options for experiments that are specified directly to the *run_experiment* command rather than in a configuration file. For example, if you would like to run an ablation experiment, which conducts repeated experiments using different combinations of the features in your config, you should use the `run_experiment --ablation` option. A complete list of options is available *here*.

Next, we describe the numerous file formats that SKLL supports for reading in features.

1.4.2 Feature files

SKLL supports the following feature file formats:

arff

The same file format used by Weka with the following added restrictions:

- Only simple numeric, string, and nominal values are supported.
- Nominal values are converted to strings.
- If the data has instance IDs, there should be an attribute with the name specified by *id_col* in the *Input* section of the configuration file you create for your experiment. This defaults to `id`. If there is no such attribute, IDs will be generated automatically.
- If the data is labelled, there must be an attribute with the name specified by *label_col* in the *Input* section of the configuartion file you create for your experiment. This defaults to `y`. This must also be the final attribute listed (like in Weka).

csv/tsv

A simple comma or tab-delimited format. SKLL underlyingly uses pandas to read these files which is extremely fast but at the cost of some extra memory consumption.

When using this file format, the following restrictions apply:

- If the data is labelled, there must be a column with the name specified by *label_col* in the *Input* section of the configuration file you create for your experiment. This defaults to `y`.
- If the data has instance IDs, there should be a column with the name specified by *id_col* in the *Input* section of the configuration file you create for your experiment. This defaults to `id`. If there is no such column, IDs will be generated automatically.
- All other columns contain feature values, and every feature value must be specified (making this a poor choice for sparse data).

Warning:

1. SKLL will raise an error if there are blank values in **any** of the columns. You must either drop all rows with blank values in any column or replace the blanks with a value you specify. To drop or replace via the command line, use the *filter_features* script. You can also drop/replace via the SKLL Reader API, specifically *`skll.data.readers.CSVReader`* and *`skll.data.readers.TSVReader`*.
2. Dropping blanks will drop **all** rows with blanks in **any** of the columns. If you care only about **some** of the columns in the file and do not want to rows to be dropped due to blanks

in the other columns, you should remove the columns you do not care about before dropping the blanks. For example, consider a hypothetical file `in.csv` that contains feature columns named `A` through `G` with the IDs stored in a column named `ID` and the labels stored in a column named `CLASS`. You only care about columns `A`, `C`, and `F` and want to drop all rows in the file that have blanks in any of these 3 columns but **do not** want to lose data due to there being blanks in any of the other columns. On the command line, you can run the following two commands:

```
$ filter_features -f A C F --id_col ID --label_col class␣
↪in.csv temp.csv
$ filter_features --id_col ID --label_col CLASS --drop_
↪blanks temp.csv out.csv
```

If you are using the SKLL Reader API, you can accomplish the same in a single step by also passing using the keyword argument `pandas_kwargs` when instantiating either a *skll.data.readers.CSVReader* or a *skll.data.readers.TSVReader*. For our example:

```
r = CSVReader.for_path('/path/to/in.csv',
                       label_col='CLASS',
                       id_col='ID',
                       drop_blanks=True,
                       pandas_kwargs={'usecols': ['A', 'C
↪', 'F', 'ID', 'CLASS']})
fs = r.read()
```

Make sure to include the ID and label columns in the *usecols* list otherwise `pandas` will drop them too.

jsonlines/ndj *(Recommended)*

A twist on the JSON format where every line is a either JSON dictionary (the entire contents of a normal JSON file), or a comment line starting with `//`. Each dictionary is expected to contain the following keys:

- **y**: The class label.
- **x**: A dictionary of feature values.
- **id**: An optional instance ID.

This is the preferred file format for SKLL, as it is sparse and can be slightly faster to load than other formats.

libsvm

While we can process the standard input file format supported by LibSVM, LibLinear, and SVM-Light, we also support specifying extra metadata usually missing from the format in comments at the of each line. The comments are not mandatory, but without them, your labels and features will not have names. The comment is structured as follows:

```
ID | 1=ClassX | 1=FeatureA 2=FeatureB
```

The entire format would like this:

```
2 1:2.0 3:8.1 # Example1 | 2=ClassY | 1=FeatureA 3=FeatureC
1 5:7.0 6:19.1 # Example2 | 1=ClassX | 5=FeatureE 6=FeatureF
```

Note: IDs, labels, and feature names cannot contain the following characters: `| # =`

1.4.3 Configuration file fields

The experiment configuration files that `run_experiment` accepts are standard Python configuration files that are similar in format to Windows INI files.[1] There are four expected sections in a configuration file: *General*, *Input*, *Tuning*, and *Output*. A detailed description of each field in each section is provided below, but to summarize:

- If you want to do **cross-validation**, specify a path to training feature files, and set *task* to `cross_validate`. Please note that the cross-validation currently uses StratifiedKFold. You also can optionally use predetermined folds with the *folds_file* setting.

  Note: When using classifiers, SKLL will automatically reduce the number of cross-validation folds to be the same as the minimum number of examples for any of the classes in the training data.

- If you want to **train a model and evaluate it** on some data, specify a training location, a test location, and a directory to store results, and set *task* to `evaluate`.
- If you want to just **train a model and generate predictions**, specify a training location, a test location, and set *task* to `predict`.
- If you want to just **train a model**, specify a training location, and set *task* to `train`.

[1] We are considering adding support for YAML configuration files in the future, but we have not added this functionality yet.

- If you want to **generate learning curves** for your data, specify a training location and set *task* to `learning_curve`. The learning curves are generated using essentially the same underlying process as in scikit-learn except that the SKLL feature pre-processing pipeline is used while training the various models and computing the scores.

  Note:

  1. Ideally, one would first do cross-validation experiments with grid search and/or ablation and get a well-performing set of features and hyper-parameters for a set of learners. Then, one would explicitly specify those features (via *featuresets*) and hyper-parameters (via *fixed_parameters*) in the config file for the learning curve and explore the impact of the size of the training data.
  2. To ensure reliable results, SKLL expects a minimum of 500 examples in the training set when generating learning curves.
  3. If you set *probability* to `True`, the probabilities will be converted to the most likely label via an `argmax` before computing the curve.

- A *list of classifiers/regressors* to try on your feature files is required.

Example configuration files are available here under the `california`, `iris`, and `titanic` subdirectories.

General

Both fields in the General section are required.

experiment_name

A string used to identify this particular experiment configuration. When generating result summary files, this name helps prevent overwriting previous summaries.

task

What types of experiment we're trying to run. Valid options are: *cross_validate*, *evaluate*, *predict*, *train*, *learning_curve*.

Input

The Input section must specify the machine learners to use via the *learners* field as well as the data and features to be used when training the model. This can be done by specifying either (a) *train_file* in which case all of the features in the file will be used, or (b) *train_directory* along with *featuresets*.

learners

List of `scikit-learn` models to be used in the experiment. Acceptable values are described below. Custom learners can also be specified. See *custom_learner_path*.

Classifiers:

- **AdaBoostClassifier**: AdaBoost Classification. Note that the default base estimator is a `DecisionTreeClassifier`. A different base estimator can be used by specifying an `estimator` fixed parameter in the *fixed_parameters* list. The following additional base estimators are supported: `MultinomialNB`, `SGDClassifier`, and `SVC`. Note that the last two base estimators require setting an additional `algorithm` fixed parameter with the value `'SAMME'`.
- **BaggingClassifier**: Bagging Classification. Note that the default base estimator is a `DecisionTreeClassifier`. A different base estimator can be used by specifying an `estimator` fixed parameter in the *fixed_parameters* list. The following additional base estimators are supported: `MultinomialNB`, `SGDClassifier`, and `SVC`. Note that when using `SVC` base estimators, you may encounter errors if you have rare classes in your data.
- **DummyClassifier**: Simple rule-based Classification
- **DecisionTreeClassifier**: Decision Tree Classification
- **GradientBoostingClassifier**: Gradient Boosting Classification
- **HistGradientBoostingClassifier**: Histogram-based Gradient Boosting Classifier. Requires dense feature array; sparse features will be automatically converted to dense when using this learner.
- **KNeighborsClassifier**: K-Nearest Neighbors Classification
- **LinearSVC**: Support Vector Classification using LibLinear
- **LogisticRegression**: Logistic Regression Classification using LibLinear
- **MLPClassifier**: Multi-layer Perceptron Classification
- **MultinomialNB**: Multinomial Naive Bayes Classification
- **RandomForestClassifier**: Random Forest Classification
- **RidgeClassifier**: Classification using Ridge Regression
- **SGDClassifier**: Stochastic Gradient Descent Classification

- **SVC**: Support Vector Classification using LibSVM
- **VotingClassifier**: Soft Voting/Majority Rule classifier for unfitted estimators. Using this learner requires specifying the underlying estimators using the `estimator_names` fixed parameter in the *fixed_parameters* list. By default, this learner uses "hard" voting, i.e., majority rule. To use "soft" voting, i.e., based on the argmax of the sums of the probabilities from the underlying classifiers, specify the `voting_type` fixed_parameter and set it to "soft". The following additional fixed parameters can also be supplied in the *fixed_parameters* list:
  - `estimator_fixed_parameters` which takes a list of dictionaries to fix any parameters in the underlying learners to desired values,
  - `estimator_param_grids` which takes a list of dictionaries specifying the possible list of parameters to search for every underlying learner,
  - `estimator_sampler_list` which can be used to specify any feature sampling algorithms for the underlying learners, and
  - `estimator_sampler_parameters` which can be used to specify any additional parameters for any specified samplers.

  Refer to this example voting configuration file to see how these parameters are used.

Regressors:

- **AdaBoostRegressor**: AdaBoost Regression. Note that the default base estimator is a `DecisionTreeRegressor`. A different base estimator can be used by specifying an `estimator` fixed parameter in the *fixed_parameters* list. The following additional base estimators are supported: `LinearRegression`, `SGDRegressor`, and `SVR`.
- **BaggingRegressor**: Bagging Regression. Note that the default base estimator is a `DecisionTreeRegressor`. A different base estimator can be used by specifying an `estimator` fixed parameter in the *fixed_parameters* list. The following additional base estimators are supported: `LinearRegression`, `SGDRegressor`, and `SVR`.
- **BayesianRidge**: Bayesian Ridge Regression. Requires dense feature array; sparse features will be automatically converted to dense when using this learner.
- **DecisionTreeRegressor**: Decision Tree Regressor
- **DummyRegressor**: Simple Rule-based Regression
- **ElasticNet**: ElasticNet Regression
- **GradientBoostingRegressor**: Gradient Boosting Regressor
- **HistGradientBoostingRegressor**: Histogram-based Gradient Boosting Regressor. Requires dense feature array; sparse features will be automatically converted to dense when using this learner.
- **HuberRegressor**: Huber Regression

- **KNeighborsRegressor**: K-Nearest Neighbors Regression
- **Lars**: Least Angle Regression. Requires dense feature array; sparse features will be automatically converted to dense when using this learner.
- **Lasso**: Lasso Regression
- **LinearRegression**: Linear Regression
- **LinearSVR**: Support Vector Regression using LibLinear
- **MLPRegressor**: Multi-layer Perceptron Regression
- **RandomForestRegressor**: Random Forest Regression
- **RANSACRegressor**: RANdom SAmple Consensus Regression. Note that the default base estimator is a `LinearRegression`. A different base regressor can be used by specifying a `estimator` fixed parameter in the *fixed_parameters* list. The following additional base estimators are supported: `LinearRegression`, `SGDRegressor`, and `SVR`.
- **Ridge**: Ridge Regression
- **SGDRegressor**: Stochastic Gradient Descent Regression
- **SVR**: Support Vector Regression using LibSVM
- **TheilSenRegressor**: Theil-Sen Regression. Requires dense feature array; sparse features will be automatically converted to dense when using this learner.
- **VotingRegressor**: Prediction voting regressor for unfitted estimators. Using this learner requires specifying the underlying estimators using the `estimator_names` fixed parameter in the *fixed_parameters* list. The following additional fixed parameters can also be supplied in this list:
  - `estimator_fixed_parameters` which takes a list of dictionaries to fix any parameters in the underlying learners to desired values,
  - `estimator_param_grids` which takes a list of dictionaries specifying the possible list of parameters to search for every underlying learner,
  - `estimator_sampler_list` which can be used to specify any feature sampling algorithms for the underlying learners, and
  - `estimator_sampler_parameters` which can be used to specify any additional parameters for any specified samplers.

  Refer to this example voting configuration file to see how these parameters are used.

For all regressors *except* `VotingRegressor`, you can also prepend `Rescaled` to the beginning of the full name (e.g., `RescaledSVR`) to get a version of the regressor where predictions are rescaled and constrained to better match the training set. Rescaled regressors can, however, be used as underlying estimators for `VotingRegressor` learners.

featuresets

List of lists of prefixes for the files containing the features you would like to train/test on. Each list will end up being a job. IDs are required to be the same in all of the feature files, and a `ValueError` will be raised if this is not the case. Cannot be used in combination with *train_file* or *test_file*.

Note: If specifying *train_directory* or *test_directory*, *featuresets* is required.

train_file

Path to a file containing the features to train on. Cannot be used in combination with *featuresets*, *train_directory*, or *test_directory*.

Note: If *train_file* is not specified, *train_directory* must be.

train_directory

Path to directory containing training data files. There must be a file for each featureset. Cannot be used in combination with *train_file* or *test_file*.

Note: If *train_directory* is not specified, *train_file* must be.

The following is a list of the other optional fields in this section in alphabetical order.

class_map *(Optional)*

If you would like to collapse several labels into one, or otherwise modify your labels (without modifying your original feature files), you can specify a dictionary mapping from new class labels to lists of original class labels. For example, if you wanted to collapse the labels `beagle` and `dachsund` into a `dog` class, you would specify the following for `class_map`:

```
{'dog': ['beagle', 'dachsund']}
```

Any labels not included in the dictionary will be left untouched.

One other use case for `class_map` is to deal with classification labels that would be converted to `float` improperly. All `Reader` sub-classes use the `skll.data.readers.safe_float` function internally to read labels. This function tries to convert a single label first to `int`, then to `float`. If neither conversion is possible, the label remains a `str`. Thus, care must be taken to ensure that labels do not get converted in unexpected ways. For example, consider the situation where there are classification labels that are a mixture of `int`-converting and `float`-converting labels:

```
import numpy as np
from skll.data.readers import safe_float
np.array([safe_float(x) for x in ["2", "2.2", "2.21"]]) # array([2.  , 2.
↪2 , 2.21])
```

The labels will all be converted to floats and any classification model generated with this data will predict labels such as `2.0`, `2.2`, etc., not `str` values that exactly match the input labels, as might be expected. `class_map` could be used to map the original labels to new values that do not have the same characteristics.

custom_learner_path *(Optional)*

Path to a `.py` file that defines a custom learner. This file will be imported dynamically. This is only required if a custom learner is specified in the list of *learners*.

All Custom learners must implement the `fit` and `predict` methods. Custom classifiers must either (a) inherit from an existing `scikit-learn` classifier, or (b) inherit from both sklearn.base.BaseEstimator. *and* from sklearn.base.ClassifierMixin.

Similarly, Custom regressors must either (a) inherit from an existing `scikit-learn` regressor, or (b) inherit from both sklearn.base.BaseEstimator. *and* from sklearn.base.RegressorMixin.

Learners that require dense matrices should implement a method `requires_dense` that returns `True`.

custom_metric_path *(Optional)*

Path to a `.py` file that defines a *custom metric function*. This file will be imported dynamically. This is only required if a custom metric is specified as a *tuning objective*, an *output metric*, or both.

cv_seed *(Optional)*

The seed to use during the creation of the folds for the *cross_validate* task. This option may be useful for running the same cross validation experiment multiple times (with the same number of differently constituted folds) to get a sense of the variance across replicates.

Note that this seed is only used for shuffling the data before splitting it into folds. The shuffling happens automatically when doing *grid search* or if *shuffle* is explicitly set to `True`. Defaults to `123456789`.

feature_hasher *(Optional)*

If `True`, this enables a high-speed, low-memory vectorizer that uses feature hashing for converting feature dictionaries into NumPy arrays instead of using a DictVectorizer. This flag will drastically reduce memory consumption for data sets with a large number of features. If enabled, the user should also specify the number of features in the *hasher_features* field. For additional information see the scikit-learn documentation.

Warning: Due to the way SKLL experiments are architected, if the features for an experiment are spread across multiple files on disk, feature hashing will be applied to each file *separately*. For example, if you have F feature files and you choose H as the number of hashed features (via *hasher_features*), you will end up with F x H features in the end. If this is not the desired behavior, use the *join_features* utility script to combine all feature files into a single file before running the experiment.

feature_scaling *(Optional)*

Whether to scale features by their mean and/or their standard deviation. If you scale by mean, your data will automatically be converted to dense, so use caution when you have a very large dataset. Valid options are:

none
Perform no feature scaling at all.

with_std
Scale feature values by their standard deviation.

with_mean
Center features by subtracting their mean.

both
Perform both centering and scaling.

Defaults to none.

featureset_names *(Optional)*

Optional list of names for the feature sets. If omitted, then the prefixes will be munged together to make names.

folds_file *(Optional)*

Path to a csv file specifying the mapping of instances in the training data to folds. This can be specified when the *task* is either `train` or `cross_validate`. For the `train` task, if *grid_search* is `True`, this file, if specified, will be used to define the cross-validation used for the grid search (leave one fold ID out at a time). Otherwise, it will be ignored.

For the `cross_validate` task, this file will be used to define the outer cross-validation loop and, if *grid_search* is `True`, also for the inner grid-search cross-validation loop. If the goal of specifiying the folds file is to ensure that the model does not learn to differentiate based on a confound: e.g. the data from the same person is always in the same fold, it makes sense to keep the same folds for both the outer and the inner cross-validation loops.

However, sometimes the goal of specifying the folds file is simply for the purpose of comparison to another existing experiment or another context in which maintaining the constitution of the folds in the inner grid-search loop is not required. In this case, users may set the parameter *use_folds_file_for_grid_search* to `False` which will then direct the inner grid-search cross-validation loop to simply use the number specified via *grid_search_folds* instead of using the folds file. This will likely lead to shorter execution times as well depending on how many folds are in the folds file and the value of *grid_search_folds*.

The format of this file must be as follows: the first row must be a header. This header row is ignored, so it doesn't matter what the header row contains, but it must be there. If there is no header row, whatever row is in its place will be ignored. The first column should consist of training set IDs and the second should be a string for the fold ID (e.g., 1 through 5, A through D, etc.). If specified, the CV and grid search will leave one fold ID out at a time.[2]

fixed_parameters *(Optional)*

List of dictionaries containing parameters you want to have fixed for each learner in the *learners* list. Empty dictionaries will be ignored and the defaults will be used for these learners. If *grid_search* is `True`, there is a potential for conflict with specified/default parameter grids and fixed parameters.

Note: Tuples are not supported in the config file, and will lead to parsing errors. Make sure to replace tuples with lists when specifying fixed parameters. As an example, consider the following parameter that's usually defined as a tuple in scikit-learn:

[2] K-1 folds will be used for grid search within CV, so there should be at least 3 fold IDs.

```
{'hidden_layer_sizes': (28, 28)}
```

To specify it in *fixed_parameters*, use a list instead:

```
{'hidden_layer_sizes': [28, 28]}
```

The default fixed parameters (beyond those that `scikit-learn` sets) are:

AdaBoostClassifier and AdaBoostRegressor

```
{'n_estimators': 500, 'random_state': 123456789}
```

BaggingClassifier and BaggingRegressor

```
{'n_estimators': 500, 'random_state': 123456789}
```

DecisionTreeClassifier and DecisionTreeRegressor

```
{'random_state': 123456789}
```

DummyClassifier

```
{'random_state': 123456789}
```

ElasticNet

```
{'random_state': 123456789}
```

GradientBoostingClassifier and GradientBoostingRegressor

```
{'n_estimators': 500, 'random_state': 123456789}
```

HistGradientBoostingClassifier and HistGradientBoostingRegressor

```
{'random_state': 123456789}
```

Lasso:

```
{'random_state': 123456789}
```

LinearSVC and LinearSVR

```
{'random_state': 123456789}
```

LogisticRegression

```
{'max_iter': 1000, 'multi_class': 'auto', 'random_state': 123456789,
↪'solver': 'liblinear'}
```

Note: The regularization `penalty` used by default is `"l2"`. However, `"l1"`, `"elasticnet"`, and `"none"` (no regularization) are also available. There is a dependency between the `penalty` and the `solver`. For example, the `"elasticnet"` penalty can *only* be used in conjunction with the `"saga"` solver. See more information in the `scikit-learn` documentation here.

MLPClassifier and MLPRegressor:

```
{'learning_rate': 'invscaling', 'max_iter': 500}
```

RandomForestClassifier and RandomForestRegressor

```
{'n_estimators': 500, 'random_state': 123456789}
```

RANSACRegressor

```
{'loss': 'squared_error', 'random_state': 123456789}
```

Ridge and RidgeClassifier

```
{'random_state': 123456789}
```

SVC and SVR

```
{'cache_size': 1000, 'gamma': 'scale'}
```

SGDClassifier

```
{'loss': 'log', 'max_iter': 1000, 'random_state': 123456789, 'tol':␣
↪1e-3}
```

SGDRegressor

```
{'max_iter': 1000, 'random_state': 123456789, 'tol': 1e-3}
```

TheilSenRegressor

```
{'random_state': 123456789}
```

Note:

The *fixed_parameters* field offers us a way to deal with imbalanced data sets by using the parameter `class_weight` for the following classifiers: `DecisionTreeClassifier`, `LogisticRegression`, `LinearSVC`, `RandomForestClassifier`, `RidgeClassifier`, `SGDClassifier`, and `SVC`.

Two possible options are available. The first one is `balanced`, which automatically adjusts weights inversely proportional to class frequencies, as shown in the following code:

```
{'class_weight': 'balanced'}
```

The second option allows you to assign a specific weight per each class. The default weight per class is 1. For example:

```
{'class_weight': {1: 10}}
```

Additional examples and information can be seen here.

hasher_features *(Optional)*

The number of features used by the FeatureHasher if the *feature_hasher* flag is enabled.

Note: To avoid collisions, you should always use the power of two larger than the number of features in the data set for this setting. For example, if you had 17 features, you would want to set the flag to 32.

id_col *(Optional)*

If you're using *ARFF*, *CSV*, or *TSV* files, the IDs for each instance are assumed to be in a column with this name. If no column with this name is found, the IDs are generated automatically. Defaults to `id`.

ids_to_floats *(Optional)*

If you have a dataset with lots of examples, and your input files have IDs that look like numbers (can be converted by float()), then setting this to True will save you some memory by storing IDs as floats. Note that this will cause IDs to be printed as floats in prediction files (e.g., `4.0` instead of `4` or `0004` or `4.000`).

label_col *(Optional)*

If you're using *ARFF*, *CSV*, or *TSV* files, the class labels for each instance are assumed to be in a column with this name. If no column with this name is found, the data is assumed to be unlabelled. Defaults to `y`. For ARFF files only, this must also be the final column to count as the label (for compatibility with Weka).

learning_curve_cv_folds_list *(Optional)*

List of integers specifying the number of folds to use for cross-validation at each point of the learning curve (training size), one per learner. For example, specifying `["SVC", "LogisticRegression"]` for `learners` and specifying `[10, 100]` for `learning_curve_cv_folds_list` will tell SKLL to use 10 cross-validation folds at each point of the SVC curve and 100 cross-validation folds at each point of the logistic regression curve. Although more folds will generally yield more reliable results, smaller number of folds may be better for learners that are slow to train. Defaults to `10` for each learner.

learning_curve_train_sizes *(Optional)*

List of floats or integers representing relative or absolute numbers of training examples that will be used to generate the learning curve of training examples that will be used to generate the learning curve respectively. If the type is float, it is regarded as a fraction of the maximum size of the training set (that is determined by the selected validation method), i.e. it has to be within (0, 1]. Otherwise it is interpreted as absolute sizes of the training sets. Note that for classification the number of samples usually has to be big enough to contain at least one sample from each class. Defaults to `[0.1, 0.325, 0.55, 0.775, 1.0]`.

num_cv_folds *(Optional)*

The number of folds to use for cross validation. Defaults to 10.

random_folds *(Optional)*

Whether to use random folds for cross-validation. Defaults to `False`.

sampler *(Optional)*

Whether to use a feature sampler that performs non-linear transformations of the input, which can serve as a basis for linear classification or other algorithms. Valid options are: Nystroem, RBFSampler, SkewedChi2Sampler, and AdditiveChi2Sampler. For additional information see the scikit-learn documentation.

Note: Using a feature sampler with the `MultinomialNB` learner is not allowed since it cannot handle negative feature values.

sampler_parameters *(Optional)*

dict containing parameters you want to have fixed for the `sampler`. Any empty ones will be ignored (and the defaults will be used).

The default fixed parameters (beyond those that `scikit-learn` sets) are:

Nystroem

```
{'random_state': 123456789}
```

RBFSampler

```
{'random_state': 123456789}
```

SkewedChi2Sampler

```
{'random_state': 123456789}
```

shuffle *(Optional)*

If `True`, shuffle the examples in the training data before using them for learning. This happens automatically when doing a grid search but it might be useful in other scenarios as well, e.g., online learning. Defaults to `False`.

suffix *(Optional)*

The file format the training/test files are in. Valid option are *.arff*, *.csv*, *.jsonlines*, *.libsvm*, *.ndj*, and *.tsv*.

If you omit this field, it is assumed that the "prefixes" listed in *featuresets* are actually complete filenames. This can be useful if you have feature files that are all in different formats that you would like to combine.

test_file *(Optional)*

Path to a file containing the features to test on. Cannot be used in combination with *featuresets*, *train_directory*, or *test_directory*

test_directory *(Optional)*

Path to directory containing test data files. There must be a file for each featureset. Cannot be used in combination with *train_file* or *test_file*.

Tuning

Generally, in this section, you would specify fields that pertain to the hyperparameter tuning for each learner. The most common required field is *objectives* although it may also be optional in certain circumstances.

objectives

A list of one or more metrics to use as objective functions for tuning the learner hyperparameters via grid search. Note that `objectives` is required by default in most cases unless (a) *grid_search* is explicitly set to `False` or (b) the task is *learning_curve*. For (a), any specified objectives are ignored. For (b), specifying objectives will raise an exception.

SKLL provides the following metrics but you can also write your own *custom metrics*.

Classification: The following objectives can be used for classification problems although some are restricted by problem type (binary/multiclass), types of labels (integers/floats/strings), and whether they are contiguous (if integers). Please read carefully.

Note: When doing classification, SKLL internally sorts and maps all the class labels in the data and maps them to integers which can be thought of class indices. This happens irrespective of the data type of the original labels. For example, if your data has the labels `['A', 'B', 'C']`, SKLL will map them to the indices `[0, 1, 2]` respectively. It will do the same if you have integer labels (`[1, 2, 3]`) or floating point ones (`[1.0, 1.1, 1.2]`). All of the tuning objectives are computed using these integer indices rather than the original class labels. This is why some metrics *only* make sense in certain scenarios. For example, SKLL only allows using weighted kappa metrics as tuning objectives if the original class labels are contiguous integers, e.g., `[1, 2, 3]` or `[4, 5, 6]` – or even integer-like floats (e,g., `[1.0, 2.0, 3.0]`, but not `[1.0, 1.1, 1.2]`).

- **accuracy**: Overall accuracy
- **average_precision**: Area under PR curve . To use this metric, *probability* must be set to `True`. (*Binary classification only*).
- **balanced_accuracy**: A version of accuracy specifically designed for imbalanced binary and multi-class scenarios.
- **f1**: The default `scikit-learn` F_1 score (F_1 of the positive class for binary classification, or the weighted average F_1 for multiclass classification)
- **f1_score_macro**: Macro-averaged F_1 score
- **f1_score_micro**: Micro-averaged F_1 score
- **f1_score_weighted**: Weighted average F_1 score
- **f1_score_least_frequent**: F_1 score of the least frequent class. The least frequent class may vary from fold to fold for certain data distributions.
- **f05**: The default `scikit-learn` $F_{=0.5}$ score ($F_{=0.5}$ of the positive class for binary classification, or the weighted average $F_{=0.5}$ for multiclass classification)
- **f05_score_macro**: Macro-averaged $F_{=0.5}$ score
- **f05_score_micro**: Micro-averaged $F_{=0.5}$ score
- **f05_score_weighted**: Weighted average $F_{=0.5}$ score
- **jaccard**: The default Jaccard similarity coefficient from `scikit-learn` for binary classification.
- **jaccard_macro**: Macro-averaged Jaccard similarity coefficient
- **jaccard_micro**: Micro-averaged Jaccard similarity coefficient
- **jaccard_weighted**: Weighted average Jaccard similarity coefficient
- **kendall_tau**: Kendall's tau . For binary classification and with *probability* set to `True`, the probabilities for the positive class will be used to compute the correlation values. In all other cases, the labels are used. (*Integer labels only*).
- **linear_weighted_kappa**: Linear weighted kappa. (*Contiguous integer labels only*).

- **lwk_off_by_one**: Same as `linear_weighted_kappa`, but all ranking differences are discounted by one. (*Contiguous integer labels only*).
- **neg_log_loss**: The negative of the classification log loss . Since `scikit-learn` recommends using negated loss functions as scorer functions, SKLL does the same for the sake of consistency. To use this metric, *probability* must be set to `True`.
- **pearson**: Pearson correlation . For binary classification and with *probability* set to `True`, the probabilities for the positive class will be used to compute the correlation values. In all other cases, the labels are used. (*Integer labels only*).
- **precision**: Precision for binary classification
- **precision_macro**: Macro-averaged Precision
- **precision_micro**: Micro-averaged Precision
- **precision_weighted**: Weighted average Precision
- **quadratic_weighted_kappa**: Quadratic weighted kappa. (*Contiguous integer labels only*). If you wish to compute quadratic weighted kappa for continuous values, you may want to use the implementation provided by RSMTool. To do so, install the RSMTool Python package and create a *custom metric* that wraps `rsmtool.utils.quadratic_weighted_kappa`.
- **qwk_off_by_one**: Same as `quadratic_weighted_kappa`, but all ranking differences are discounted by one. (*Contiguous integer labels only*).
- **recall**: Recall for binary classification
- **recall_macro**: Macro-averaged Recall
- **recall_micro**: Micro-averaged Recall
- **recall_weighted**: Weighted average Recall
- **roc_auc**: Area under ROC curve .To use this metric, *probability* must be set to `True`. (*Binary classification only*).
- **spearman**: Spearman rank-correlation. For binary classification and with *probability* set to `True`, the probabilities for the positive class will be used to compute the correlation values. In all other cases, the labels are used. (*Integer labels only*).
- **unweighted_kappa**: Unweighted Cohen's kappa.
- **uwk_off_by_one**: Same as `unweighted_kappa`, but all ranking differences are discounted by one. In other words, a ranking of 1 and a ranking of 2 would be considered equal.

Regression: The following objectives can be used for regression problems.

- **explained_variance**: A score indicating how much of the variance in the given data can be by the model.

- **kendall_tau**: Kendall's tau
- **linear_weighted_kappa**: Linear weighted kappa (any floating point values are rounded to ints)
- **lwk_off_by_one**: Same as `linear_weighted_kappa`, but all ranking differences are discounted by one.
- **max_error**: The maximum residual error.
- **neg_mean_absolute_error**: The negative of the mean absolute error regression loss. Since `scikit-learn` recommends using negated loss functions as scorer functions, SKLL does the same for the sake of consistency.
- **neg_mean_squared_error**: The negative of the mean squared error regression loss. Since `scikit-learn` recommends using negated loss functions as scorer functions, SKLL does the same for the sake of consistency.
- **neg_root_mean_squared_error**: The negative of the mean squared error regression loss, with `squared` set to False. Since `scikit-learn` recommends using negated loss functions as scorer functions, SKLL does the same for the sake of consistency.
- **pearson**: Pearson correlation
- **quadratic_weighted_kappa**: Quadratic weighted kappa (any floating point values are rounded to ints)
- **qwk_off_by_one**: Same as `quadratic_weighted_kappa`, but all ranking differences are discounted by one.
- **r2**: R2
- **spearman**: Spearman rank-correlation
- **unweighted_kappa**: Unweighted Cohen's kappa (any floating point values are rounded to ints)
- **uwk_off_by_one**: Same as `unweighted_kappa`, but all ranking differences are discounted by one. In other words, a ranking of 1 and a ranking of 2 would be considered equal.

The following is a list of the other optional fields in this section in alphabetical order.

grid_search *(Optional)*

Whether or not to perform grid search to find optimal parameters for the learner. Defaults to `True` since optimizing model hyperparameters almost always leads to better performance. Note that for the *learning_curve* task, grid search is not allowed and setting it to `True` will generate a warning and be ignored.

Note:

1. In versions of SKLL before v2.0, this option was set to `False` by default but that was changed since the benefits of hyperparameter tuning significantly outweigh the cost in terms of model fitting time. Instead, SKLL users must explicitly opt out of hyperparameter tuning if they so desire.
2. Although SKLL only uses the combination of hyperparameters in the grid that maximizes the grid search objective, the results for all other points on the grid that were tried are also available. See the `grid_search_cv_results` attribute in the `.results.json` file.

---

grid_search_folds *(Optional)*

The number of folds to use for grid search. Defaults to 5.

grid_search_jobs *(Optional)*

Number of folds to run in parallel when using grid search. Defaults to number of grid search folds.

min_feature_count *(Optional)*

The minimum number of examples for which the value of a feature must be nonzero to be included in the model. Defaults to 1.

param_grids *(Optional)*

List of parameter grid dictionaries, one for each learner. Each parameter grid is a dictionary mapping from strings to list of parameter values. When you specify an empty dictionary for a learner, the default parameter grid for that learner will be searched.

The default parameter grids for each learner are:

AdaBoostClassifier and AdaBoostRegressor

```
{'learning_rate': [0.01, 0.1, 1.0, 10.0, 100.0]}
```

BaggingClassifier and BaggingRegressor

```
{'max_samples': [0.1, 0.25, 0.5, 1.0],
 'max_features': [0.1, 0.25, 0.5, 1.0]}
```

BayesianRidge

```
{'alpha_1': [1e-6, 1e-4, 1e-2, 1, 10],
 'alpha_2': [1e-6, 1e-4, 1e-2, 1, 10],
 'lambda_1': [1e-6, 1e-4, 1e-2, 1, 10],
 'lambda_2': [1e-6, 1e-4, 1e-2, 1, 10]}
```

DecisionTreeClassifier and DecisionTreeRegressor

```
{'max_features': ["sqrt", None]}
```

ElasticNet

```
{'alpha': [0.01, 0.1, 1.0, 10.0, 100.0]}
```

GradientBoostingClassifier and GradientBoostingRegressor

```
{'max_depth': [1, 3, 5]}
```

HistGradientBoostingClassifier

```
{'learning_rate': [0.01, 0.1, 1.0],
 'min_samples_leaf': [10, 20, 40]}
```

HistGradientBoostingRegressor

```
{'loss': ['squared_error', 'absolute_error', 'poisson'],
 'learning_rate': [0.01, 0.1, 1.0],
 'min_samples_leaf': [10, 20, 40]}
```

HuberRegressor

```
{'epsilon': [1.05, 1.35, 1.5, 2.0, 2.5, 5.0],
 'alpha': [1e-4, 1e-3, 1e-3, 1e-1, 1, 10, 100, 1000]}
```

KNeighborsClassifier and KNeighborsRegressor

```
{'n_neighbors': [1, 5, 10, 100],
 'weights': ['uniform', 'distance']}
```

Lasso

```
{'alpha': [0.01, 0.1, 1.0, 10.0, 100.0]}
```

LinearSVC

```
{'C': [0.01, 0.1, 1.0, 10.0, 100.0]}
```

LogisticRegression

```
{'C': [0.01, 0.1, 1.0, 10.0, 100.0]}
```

MLPClassifier and MLPRegressor:

```
{'activation': ['logistic', 'tanh', 'relu'],
 'alpha': [1e-4, 1e-3, 1e-3, 1e-1, 1],
 'learning_rate_init': [0.001, 0.01, 0.1]},
```

MultinomialNB

```
{'alpha': [0.1, 0.25, 0.5, 0.75, 1.0]}
```

RandomForestClassifier and RandomForestRegressor

```
{'max_depth': [1, 5, 10, None]}
```

Ridge and RidgeClassifier

```
{'alpha': [0.01, 0.1, 1.0, 10.0, 100.0]}
```

SGDClassifier and SGDRegressor

```
{'alpha': [0.000001, 0.00001, 0.0001, 0.001, 0.01],
 'penalty': ['l1', 'l2', 'elasticnet']}
```

SVC

```
{'C': [0.01, 0.1, 1.0, 10.0, 100.0],
 'gamma': ['auto', 0.01, 0.1, 1.0, 10.0, 100.0]}
```

SVR

```
{'C': [0.01, 0.1, 1.0, 10.0, 100.0]}
```

Note:

1. Learners not listed here do not have any default parameter grids in SKLL either because there are no hyper-parameters to tune or decisions about which parameters to tune (and how) depend on the data being used for the experiment and are best left up to the user.
2. Tuples are not supported in the config file, and will lead to parsing errors. Make sure to replace tuples with lists when specifying fixed parameters. As an example, consider the following parameter that's usually defined as a tuple in scikit-learn:

   ```
   {'hidden_layer_sizes': (28, 28)}
   ```

   To specify it in *param_grids*, use a list instead:

   ```
   {'hidden_layer_sizes': [28, 28]}
   ```

pos_label *(Optional)*

A string denoting the label of the class to be treated as the positive class in a binary classification setting. If unspecified, the class represented by the label that appears second when sorted is chosen as the positive class. For example, if the two labels in data are "A" and "B" and `pos_label` is not specified, "B" will be chosen as the positive class.

use_folds_file_for_grid_search *(Optional)*

Whether to use the specified *folds_file* for the inner grid-search cross-validation loop when *task* is set to `cross_validate`. Defaults to `True`.

Note: This flag is ignored for all other tasks, including the `train` task where a specified *folds_file* is *always* used for the grid search.

Output

The fields in this section generally pertain to the *output files* produced by the experiment. The most common fields are `logs`, `models`, `predictions`, and `results`. These fields are mostly optional although they may be required in certain cases. A common option is to use the same directory for all of these fields.

logs *(Optional)*

Directory to store SKLL *log files* in. If omitted, the current working directory is used.

models *(Optional)*

Directory in which to store *trained models*. Can be omitted to not store models except when using the *train* task, where this path *must* be specified. On the other hand, this path must *not* be specified for the *learning_curve* task.

metrics *(Optional)*

For the `evaluate` and `cross_validate` tasks, this is an optional list of additional metrics that will be computed *in addition to* the tuning objectives and added to the results files. However, for the *learning_curve* task, this list is **required**. Possible values are all of the same functions as those available for the *tuning objectives* (with the same caveats).

As with objectives, You can also use your own *custom metric* functions.

Note: If the list of metrics overlaps with the grid search tuning *objectives*, then, for each job, the objective that overlaps is *not* computed again as a metric. Recall that each SKLL job can only contain a single tuning objective. Therefore, if, say, the `objectives` list is `['accuracy', 'roc_auc']` and the `metrics` list is `['roc_auc', 'average_precision']`, then in the second job, `roc_auc` is used as the objective but *not* computed as an additional metric.

pipeline *(Optional)*

Whether or not the final learner object should contain a `pipeline` attribute that contains a `scikit-learn` Pipeline object composed of copies of each of the following steps of training the learner:

- feature vectorization (*vectorizer*)
- feature selection (*selector*)
- feature sampling (*sampler*)
- feature scaling (*scaler*)
- main estimator (*estimator*)

The strings in the parentheses represent the name given to each step in the pipeline.

The goal of this attribute is to allow better interoperability between SKLL learner objects and `scikit-learn`. The user can train the model in SKLL and then further tweak or analyze the pipeline in `scikit-learn`, if needed. Each component of the pipeline is a (deep) copy of the component that was fit as part of the SKLL model training process. We use copies since we do not want the original SKLL model to be affected if the user modifies the components of the pipeline in `scikit-learn` space.

Here's an example of how to use this attribute.

```
from sklearn.preprocessing import LabelEncoder

from skll.data import Reader
from skll.learner import Learner

# train a classifier and a regressor using the SKLL API
fs1 = Reader.for_path('examples/iris/train/example_iris_features.jsonlines
↪').read()
learner1 = Learner('LogisticRegression', pipeline=True)
_ = learner1.train(fs1, grid_search=True, grid_objective='f1_score_macro')

fs2 = Reader.for_path('examples/california/train/example_california_
↪features.jsonlines').read()
learner2 = Learner('RescaledSVR', feature_scaling='both', pipeline=True)
_ = learner2.train(fs2, grid_search=True, grid_objective='pearson')

# now, we can explore the stored pipelines in sklearn space
enc = LabelEncoder().fit(fs1.labels)

# first, the classifier
D1 = {"f0": 6.1, "f1": 2.8, "f2": 4.7, "f3": 1.2}
pipeline1 = learner1.pipeline
enc.inverse_transform(pipeline1.predict(D1))

# then, the regressor
D2 = {"f0": 4.1344, "f1": 36.0, "f2": 4.1, "f3": 0.98, "f4": 1245.0, "f5
↪": 3.0, "f6": 33.9, "f7": -118.32}
pipeline2 = learner2.pipeline
pipeline2.predict(D2)

# note that without the `pipeline` attribute, one would have to
# do the following for D1, which is much less readable
enc.inverse_transform(learner1.model.predict(learner1.scaler.
↪transform(learner1.feat_selector.transform(learner1.feat_vectorizer.
↪transform(D1)))))
```

Note:

1. When using a DictVectorizer in SKLL along with *feature_scaling* set to either `with_mean` or `both`, the *sparse* attribute of the vectorizer stage in the pipeline is set to `False` since centering requires dense arrays.
2. When feature hashing is used (via a FeatureHasher) in SKLL along with *feature_scaling* set to either `with_mean` or `both` , a custom pipeline stage (`skll.learner.Densifier`) is inserted in the pipeline between the feature vectorization (here, hashing) stage and the feature scaling stage. This is necessary since a `FeatureHasher` does not have a `sparse` attribute to turn off – it *only* returns sparse vectors.
3. A `Densifier` is also inserted in the pipeline when using a SkewedChi2Sampler for feature sampling since this sampler requires dense input and cannot be made to work with sparse arrays.

predictions *(Optional)*

Directory in which to store *prediction files*. Must *not* be specified for the *learning_curve* and *train* tasks. If omitted, the current working directory is used.

probability *(Optional)*

Whether or not to output probabilities for each class instead of the most probable class for each instance. Only really makes a difference when storing predictions. Defaults to `False`. Note that this also applies to the tuning objective.

results *(Optional)*

Directory in which to store *result files*. If omitted, the current working directory is used.

save_cv_folds *(Optional)*

Whether to save the *folds file* containing the folds for a cross-validation experiment. Defaults to `True`.

save_cv_models *(Optional)*

Whether to save each of the K *model files* trained during each step of a K-fold cross-validation experiment. Defaults to `False`.

save_votes *(Optional)*

Whether to save the predictions from the individual estimators underlying a `VotingClassifer` or `VotingRegressor`. Note that for this to work, *predictions* must be set. Defaults to `False`.

wandb_credentials *(Optional)*

To enable logging metrics and artifacts to Weights & Biases, specify a dictionary as follows:

```
{'wandb_entity': 'your_entity_name', 'wandb_project': 'your_project_name'}
```

`wandb_entity` can be a user name or the name of a team or organization. `wandb_project` is the name of the project to which this experiment will be logged. If a project by this name does not already exist, it will be created. For more details on what will be logged, and an example report, see *Integration with Weights & Biases*.

Important:

1. Both *wandb_entity* and *wandb_project* must be specified. If any of them is missing, logging to W&B will not be enabled.
2. Before using Weights & Biases for the first time, users should log in and provide their API key as described in W&B Quickstart guidelines.
3. Note that when using W&B logging, a SKLL run may take significantly longer due to the network traffic being sent to W&B.

1.4.4 Using run_experiment

Once you have created the *configuration file* for your experiment, you can usually just get your experiment started by running `run_experiment CONFIGFILE`.[3] That said, there are a few options that are specified via command-line arguments instead of in the configuration file:

[3] If you installed SKLL via pip on macOS, you might get an error when using `run_experiment` to generate learning curves. To get around this, add `MPLBACKEND=Agg` before the `run_experiment` command and re-run.

-a <num_features>, **--ablation** <num_features>

Runs an ablation study where repeated experiments are conducted with the specified number of feature files in each featureset in the configuration file held out. For example, if you have three feature files (`A`, `B`, and `C`) in your featureset and you specifiy `--ablation 1`, there will be three experiments conducted with the following featuresets: `[[A, B], [B, C], [A, C]]`. Additionally, since every ablation experiment includes a run with all the features as a baseline, the following featureset will also be run: `[[A, B, C]]`.

If you would like to try all possible combinations of feature files, you can use the *`run_experiment --ablation_all`* option instead.

> **Warning:** Ablation will *not* work if you specify a *train_file* and *test_file* since no featuresets are defined in that scenario.

-A, --ablation_all

Runs an ablation study where repeated experiments are conducted with all combinations of feature files in each featureset.

> **Warning:** This can create a huge number of jobs, so please use with caution.

-k, --keep-models

If trained models already exist for any of the learner/featureset combinations in your configuration file, just load those models and do not retrain/overwrite them.

-r, --resume

If result files already exist for an experiment, do not overwrite them. This is very useful when doing a large ablation experiment and part of it crashes.

-v, --verbose

Print more status information. For every additional time this flag is specified, output gets more verbose.

--version

Show program's version number and exit.

GridMap options

If you have GridMap installed, **`run_experiment`** will automatically schedule jobs on your DRMAA- compatible cluster. You can use the following options to customize this behavior.

-l, --local

Run jobs locally instead of using the cluster.[4]

[4] This will happen automatically if GridMap cannot be imported.

-q <queue>, **--queue** <queue>

Use this queue for GridMap. (default: `all.q`)

-m <machines>, **--machines** <machines>

Comma-separated list of machines to add to GridMap's whitelist. If not specified, all available machines are used.

Note: Full names must be specified, (e.g., `nlp.research.ets.org`).

1.4.5 Output files

For most of the SKLL tasks the various output files generated by *run_experiment* share the automatically generated prefix `<EXPERIMENT>_<FEATURESET>_<LEARNER>_<OBJECTIVE>`, where the following definitions hold:

`<EXPERIMENT>`
The value of the *experiment_name* field in the configuration file.

`<FEATURESET>`
The components of the feature set that was used for training, joined with "+".

`<LEARNER>`
The learner that was used to generate the current results/model/etc.

`<OBJECTIVE>`
The objective function that was used to generate the current results/model/etc.

Note: In SKLL terminology, a specific combination of featuresets, learners, and objectives specified in the configuration file is called a `job`. Therefore, an experiment (represented by a configuration file) can contain multiple jobs.

However, if the *objectives* field in the configuration file contains only a single value, the job can be disambiguated using only the featuresets and the learners since the objective is fixed. Therefore, the output files will have the prefix `<EXPERIMENT>_<FEATURESET>_<LEARNER>`. Similarly, if a task has a single *feature set*, the output files prefix will not include the `<FEATURESET>` component.

The following types of output files can be generated after running an experiment configuration file through *run_experiment*. Note that some file types may or may not be generated depending on the values of the fields specified in the *Output section* of the configuration file.

Log files

SKLL produces two types of log files – one for each job in the experiment and a single, top level log file for the entire experiment. Each of the job log files have the usual job prefix as described above whereas the experiment log file is simply named `<EXPERIMENT>.log`.

While the job-level log files contain messages that pertain to the specific characteristics of the job (e.g., warnings from `scikit-learn` pertaining to the specific learner), the experiment-level log file will contain logging messages that pertain to the overall experiment and configuration file (e.g., an incorrect option specified in the configuration file). The messages in all SKLL log files are in the following format:

```
<TIMESTAMP> - <LEVEL> - <MSG>
```

where `<TIMESTAMP>` refers to the exact time when the message was logged, `<LEVEL>` refers to the level of the logging message (e.g., `INFO`, `WARNING`, etc.), and `<MSG>` is the actual content of the message. All of the messages are also printed to the console in addition to being saved in the job-level log files and the experiment-level log file.

Model files

Model files end in `.model` and are serialized *`skll.learner.Learner`* instances. *run_experiment* will re-use existing model files if they exist, unless it is explicitly told not to. These model files can also be loaded programmatically via the SKLL API, specifically the *`skll.learner.Learner.from_file()`* method.

Results files

SKLL generates two types of result files:

1. Files ending in `.results` which contain a human-readable summary of the job, complete with confusion matrix, objective function score on the test set, and values of any additional metrics specified via the *metrics* configuration file option.

2. Files ending in `.results.json`, which contain all of the same information as the `.results` files, but in a format more well-suited to automated processing. In some cases, `.results.json` files may contain *more* information than their `.results` file counterparts. For example, when doing *grid search* for tuning model hyperparameters, these files contain an additional attribute `grid_search_cv_results` containing detailed results from the grid search process.

Prediction files

Predictions files are TSV files that contain either the predicted values (for regression) OR predicted labels/class probabiltiies (for classification) for each instance in the test feature set. The value of the *probability* option decides whether SKLL outputs the labels or the probabilities.

When the predictions are labels or values, there are only two columns in the file: one containing the ID for the instance and the other containing the prediction. The headers for the two columns in this case are "id" and "prediction".

When the predictions are class probabilities, there are N+1 columns in these files, where N is the number of classes in the training data. The header for the column containing IDs is still "id" and the labels themselves are the headers for the columns containing their respective probabilities. In the special case of binary classification, the *positive class* probabilities are always in the last column.

Summary file

For every experiment you run, there will also be an experiment summary file generated that is a tab-delimited file summarizing the results for each job in the experiment. It is named `<EXPERIMENT>_summary.tsv`. For *learning_curve* experiments, this summary file will contain training set sizes and the averaged scores for all combinations of featuresets, learners, and objectives.

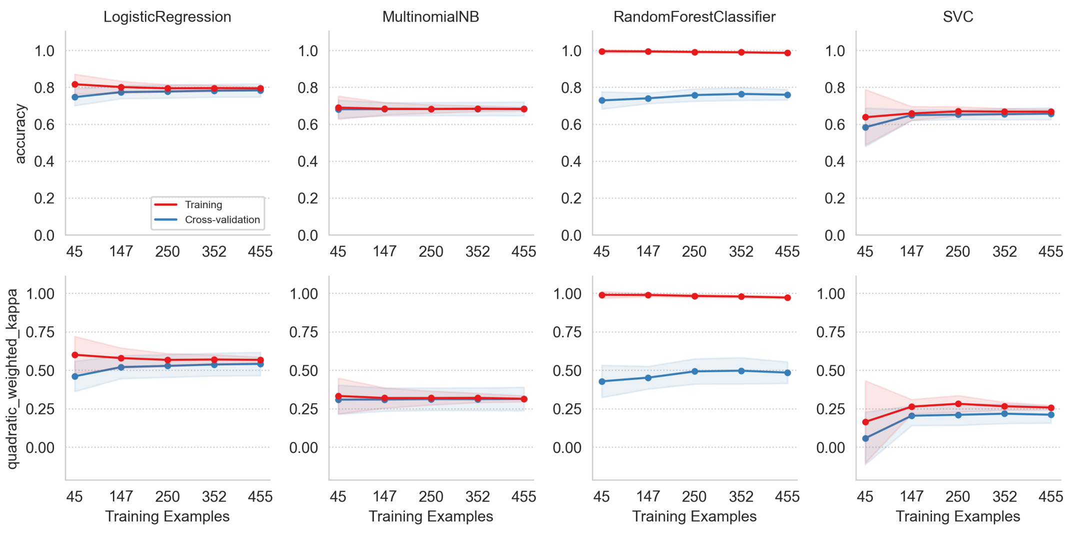
Folds file

For the *cross_validate* task, SKLL can also output the actual folds and instance IDs used in the cross-validation process, if the *save_cv_folds* option is enabled. In this case, a file called `<EXPERIMENT>_skll_fold_ids.csv` is saved to disk.

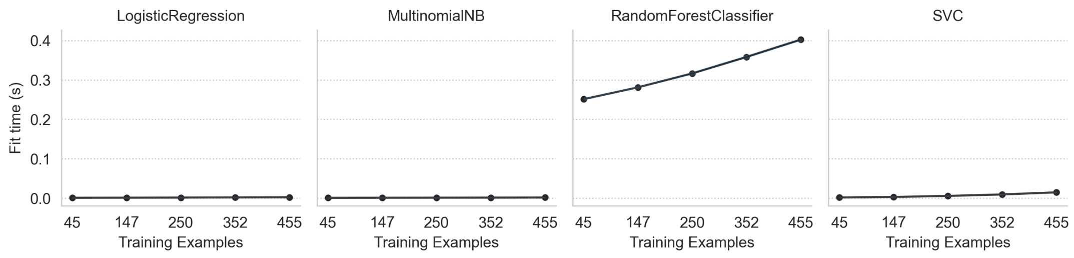
Learning curve plots

When running a *learning_curve* experiment, actual learning curves are also generated as `.png` files. Two curves are generated for each feature set specified in the configuration file.

The first `.png` file is named `EXPERIMENT_FEATURESET.png` and contains a double-faceted learning curve plot for the featureset with the specified *output metrics* along the rows and the *learners* along the columns. Each sub-plot has the number of training examples on the x-axis and the metric score on the y-axis. Here's an example of such a plot.



The second `.png` file is named `EXPERIMENT_FEATURESET_times.png` and contains a column-faceted learning curve plot for the featureset with a single row and the specified *learners* along the columns. Each sub-plot has the number of training examples on the x-axis and the model fit times on the y-axis. Here's an example of this plot.



You can also generate the plots from the learning curve summary file using the *plot_learning_curves* utility script.

1.4.6 Integration with Weights & Biases

The output of any SKLL experiment can be automatically logged to Weights & Biases. Once the logging is *enabled*, a new run will be created under the specified W&B project. The following is logged for *all* tasks:

- The SKLL configuration file, including default values for fields that were left unspecified
- The learner, feature set, and size of training and testing sets for each job in the experiment

There are additional items logged depending on the task type:

- **train**: The full path to the generated model file is logged in the project summary.
- **predict**: The predictions file is logged as a table, separately for each job in the experiment.

- **evaluate**: The task summary file is logged as a table. For classification experiments, the confusion matrix as well as a table that shows per-label precision, recall and f-measure are logged for each job.
- **cross_validate**: Similar output logged as the *evaluate* task, with a separate job per CV fold.
- **learning_curve** The summary file is logged as a table, and all learning curve plots are logged as media artifacts.

The above information logged to Weights & Biases can then be used to create informative reports for your SKLL experiments. As an example, here is a report created on Weights & Biases, based on the data logged while running the *titanic tutorial*. The report contains three sections, one per SKLL task, with a subset of the output tables and metrics that were logged for each of the tasks.

To view the full output and create your own reports, turn on *logging to Weights and Biases* in the configuration `Output` section.

1.5 Using Custom Metrics

Although SKLL comes with a huge number of built-in metrics for both classification and regression, there might be occasions when you want to use a custom metric function for hyper-parameter tuning or for evaluation. This section shows you how to do that.

1.5.1 Writing Custom Metric Functions

First, let's look at how to write valid custom metric functions. A valid custom metric function must take two array-like positional arguments: the first being the true labels or scores, and the second being the predicted labels or scores. This function can also take two optional keyword arguments:

1. `greater_is_better`: a boolean keyword argument that indicates whether a higher value of the metric indicates better performance (`True`) or vice versa (`False`). The default value is `True`.
2. `response_method` : a string keyword argument that specifies the response method to use to get predictions from an estimator. Possible values are:
   - `"predict"` : uses estimator's predict() method to get class labels
   - `"predict_proba"` : uses estimator's predict_proba() method to get class probabilities
   - `"decision_function"` : uses estimator's decision_function() method to get continuous decision function values
   - If the value is a list or tuple of the above strings, it indicates that the scorer should use the first method in the list which is implemented by the estimator.
   - If the value is `None`, it is the same as `"predict"`.

The default value for `response_method` is `None`.

Note that these keyword arguments are identical to the keyword arguments for the sklearn.metrics.make_scorer() function and serve the same purpose.

Important: Previous versions of SKLL offered the `needs_proba` and `needs_threshold` keyword arguments for custom metrics but these are now deprecated in scikit-learn and replaced by the `response_method` keyword argument. To replicate the behavior of `needs_proba=True`, use `response_method="predict_proba"` instead and to replicate `needs_threshold=True`, use `response_method=("decision_function", "predict_proba")` instead.

In short, custom metric functions take two required positional arguments (order matters) and two optional keyword arguments. Here's a simple example of a custom metric function: F with =0.75 defined in a file called `custom.py`.

Listing 1: custom.py

```python
from sklearn.metrics import fbeta_score

def f075(y_true, y_pred):
    return fbeta_score(y_true, y_pred, beta=0.75)
```

Obviously, you may write much more complex functions that aren't directly available in scikit-learn. Once you have written your metric function, the next step is to use it in your SKLL experiment.

1.5.2 Using in Configuration Files

The first way of using custom metric functions is via your SKLL experiment configuration file if you are running SKLL via the command line. To do so:

1. Add a field called *custom_metric_path* in the *Input* section of your configuration file and set its value to be the path to the `.py` file containing your custom metric function.
2. Add the name of your custom metric function to either the *objectives* field in the *Tuning* section (if you wish to use it to tune the model hyper-parameters) or to the *metrics* field in the *Output* section if you wish to only use it for evaluation. You can also add it to both.

Here's an example configuration file using data from the *SKLL Titanic example* that illustrates this. This file assumes that the file `custom.py` above is located in the same directory.

```
[General]
experiment_name = titanic
task = evaluate

[Input]
```

(continues on next page)

(continued from previous page)

```
train_directory = train
test_directory = dev
featuresets = [["family.csv", "misc.csv", "socioeconomic.csv", "vitals.csv
↪"]]
learners = ["RandomForestClassifier", "DecisionTreeClassifier", "SVC",
↪"MultinomialNB"]
label_col = Survived
id_col = PassengerId
custom_metric_path = custom.py

[Tuning]
grid_search = true
objectives = ['f075']

[Output]
metrics = ['roc_auc']
probability = true
logs = output
results = output
predictions = output
models = output
```

And that's it! SKLL will dynamically load and use your custom metric function when you *run your experiment*. Custom metric functions can be used for both hyper-parameter tuning and for evaluation.

1.5.3 Using via the API

To use a custom metric function via the SKLL API, you first need to register the custom metric function using the `register_custom_metric()` function and then just use the metric name either as a grid search objective, an output metric, or both.

Here's a short example that shows how to use the `f075()` custom metric function we defined above via the SKLL API. Again, we assume that `custom.py` is located in the current directory.

```
from skll.data import CSVReader
from skll.learner import Learner
from skll.metrics import register_custom_metric

# register the custom function with SKLL
_ = register_custom_metric("custom.py", "f075")
```

(continues on next page)

(continued from previous page)

```
# let's assume the training data lives in a file called "train.csv"
# we load that into a SKLL FeatureSet
fs = CSVReader.for_path("train.csv").read()

# instantiate a learner and tune its parameters using the custom metric
learner = Learner('LogisticRegression')
learner.train(fs, grid_objective="f075")

...
```

As with configuration files, custom metric functions can be used for both training as well as evaluation with the API.

Important:

1. When using the API, if you have multiple metric functions defined in a Python source file, you must register each one individually using `register_custom_metric()`.
2. When using the API, if you try to re-register the same metric in the same Python session, it will raise a `NameError`. Therefore, if you edit your custom metric, you must start a new Python session to be able to see the changes.
3. When using the API, if the names of any of your custom metric functions conflict with names of *metrics* that already exist in either SKLL or scikit-learn, it will raise a `NameError`. You should rename the metric function in that case.
4. When using a configuration file, if your custom metric name conflicts with names of *metrics* that already exist in either SKLL or scikit-learn, it will be silently ignored in favor of the already existing metric.
5. Unlike for the built-in metrics, SKLL does not check whether your custom metric function is appropriate for classification or regression. You must make that decision for yourself.

1.6 Utility Scripts

In addition to the main script, *run_experiment*, SKLL comes with a number of helpful utility scripts that can be used to prepare feature files and perform other routine tasks. Each is described briefly below.

1.6.1 compute_eval_from_predictions

Compute evaluation metrics from prediction files after you have run an experiment.

Positional Arguments

`examples_file`
SKLL input file with labeled examples

`predictions_file`
file with predictions from SKLL

`metric_names`
metrics to compute

Optional Arguments

`--version`
Show program's version number and exit.

1.6.2 filter_features

Filter feature file to remove (or keep) any instances with the specified IDs or labels. Can also be used to remove/keep feature columns.

Warning: Starting with v2.5 of SKLL, the arguments for `filter_features` have changed and are no longer backwards compatible with older versions of SKLL. Specifically:

1. The input and output files must now be specified with `-i` and `-o` respectively.
2. `--inverse` must now be used to invert the filtering command since `-i` is used to specify the input file.

Required Arguments

-i, --input

Input feature file (ends in `.arff`, `.csv`, `.jsonlines`, `.ndj`, or `.tsv`)

-o, --output

Output feature file (must have same extension as input file)

Optional Arguments

-f `<feature <feature ...>>`, **--feature** `<feature <feature ...>>`

A feature in the feature file you would like to keep. If unspecified, no features are removed.

-I `<id <id ...>>`, **--id** `<id <id ...>>`

An instance ID in the feature file you would like to keep. If unspecified, no instances are removed based on their IDs.

--inverse

Instead of keeping features and/or examples in lists, remove them.

--id_col `<id_col>`

Name of the column which contains the instance IDs in ARFF, CSV, or TSV files. (default: `id`)

-L `<label <label ...>>`, **--label** `<label <label ...>>`

A label in the feature file you would like to keep. If unspecified, no instances are removed based on their labels.

-l `<label_col>`, **--label_col** `<label_col>`

Name of the column which contains the class labels in ARFF, CSV, or TSV files. For ARFF files, this must be the final column to count as the label. (default: `y`)

-db, --drop-blanks

Drop all lines/rows that have any blank values. (default: `False`)

-rb `<replacement>`, **--replace-blanks-with** `<replacement>`

Specifies a new value with which to replace blank values in all columns in the file. To replace blanks differently in each column, use the SKLL Reader API directly. (default: `None`)

-q, --quiet

Suppress printing of `"Loading..."` messages.

--version

Show program's version number and exit.

1.6.3 generate_predictions

Loads a trained model and outputs predictions based on input feature files. Useful if you want to reuse a trained model as part of a larger system without creating configuration files. Offers the following modes of operation:

- For non-probabilistic classification and regression, generate the predictions.
- For probabilistic classification, generate either the most likely labels or the probabilities for each class label.
- For binary probablistic classification, generate the positive class label only if its probability exceeds the given threshold. The positive class label is either read from the model file or inferred the same way as a SKLL learner would.

Positional Arguments

`model_file`

Model file to load and use for generating predictions.

`input_file(s)`

One or more feature file(s) (ending in `.arff`, `.csv`, `.jsonlines`, `.libsvm`, `.ndj`, or `.tsv`) (with or without the label column), with the appropriate suffix.

Optional Arguments

`-i` `<id_col>`, **`--id_col`** `<id_col>`

Name of the column which contains the instance IDs in ARFF, CSV, or TSV files. (default: `id`)

`-l` `<label_col>`, **`--label_col`** `<label_col>`

Name of the column which contains the labels in ARFF, CSV, or TSV files. For ARFF files, this must be the final column to count as the label. (default: `y`)

`-o` `<path>`, **`--output_file`** `<path>`

Path to output TSV file. If not specified, predictions will be printed to stdout. For probabilistic binary classification, the probability of the positive class will always be in the last column.

`-p`, **`--predict_labels`**

If the model does probabilistic classification, output the class label with the highest probability instead of the class probabilities.

`-q`, **`--quiet`**

Suppress printing of "`Loading...`" messages.

-t <threshold>, **--threshold** <threshold>
If the model does binary probabilistic classification, return the positive class label only if it meets/exceeds the given threshold and the other class label otherwise.

--version
Show program's version number and exit.

1.6.4 join_features

Combine multiple feature files into one larger file.

Positional Arguments

infile ...
Input feature files (ends in `.arff`, `.csv`, `.jsonlines`, `.ndj`, or `.tsv`)

outfile
Output feature file (must have same extension as input file)

Optional Arguments

-l <label_col>, **--label_col** <label_col>
Name of the column which contains the labels in ARFF, CSV, or TSV files. For ARFF files, this must be the final column to count as the label. (default: `y`)

-q, **--quiet**
Suppress printing of `"Loading..."` messages.

--version
Show program's version number and exit.

1.6.5 plot_learning_curves

Generate learning curve plots from a learning curve output TSV file.

Positional Arguments

`tsv_file`

Input learning Curve TSV output file.

`output_dir`

Output directory to store the learning curve plots.

1.6.6 print_model_weights

Prints out the weights of a given trained model. If the model was trained using *feature hashing*, feature names of the form `hashed_feature_XX` will be used since the original feature names no longer apply.

Positional Arguments

`model_file`

Model file to load.

Optional Arguments

`--k` <k>

Number of top features to print (0 for all) (default: 50)

`--sign` `{positive,negative,all}`

Show only positive, only negative, or all weights (default: `all`)

`--sort_by_labels`

Order the features by classes (default: `False`). Mutually exclusive with the `--k` option.

`--version`

Show program's version number and exit.

1.6.7 skll_convert

Convert between .arff, .csv., .jsonlines, .libsvm, and .tsv formats.

Positional Arguments

`infile`

Input feature file (ends in `.arff`, `.csv`, `.jsonlines`, `.libsvm`, `.ndj`, or `.tsv`)

`outfile`

Output feature file (ends in `.arff`, `.csv`, `.jsonlines`, `.libsvm`, `.ndj`, or `.tsv`)

Optional Arguments

`-l` `<label_col>`, **`--label_col`** `<label_col>`

Name of the column which contains the labels in ARFF, CSV, or TSV files. For ARFF files, this must be the final column to count as the label. (default: `y`)

`-q`, **`--quiet`**

Suppress printing of `"Loading..."` messages.

`--arff_regression`

Create ARFF files for regression, not classification.

`--arff_relation` `ARFF_RELATION`

Relation name to use for ARFF file. (default: `skll_relation`)

`--no_labels`

Used to indicate that the input data has no labels.

`--reuse_libsvm_map` `REUSE_LIBSVM_MAP`

If you want to output multiple files that use the same mapping from labels and features to numbers when writing libsvm files, you can specify an existing .libsvm file to reuse the mapping from.

`--version`

Show program's version number and exit.

1.6.8 summarize_results

Creates an experiment summary TSV file from a list of JSON files generated by *run_experiment*.

Positional Arguments

`summary_file`
TSV file to store summary of results.

`json_file`
JSON results file generated by run_experiment.

Optional Arguments

`-a, --ablation`
The results files are from an ablation run.

`--version`
Show program's version number and exit.

1.7 API Documentation

1.7.1 Quickstart

Here is a quick run-down of how you accomplish common tasks.

Load a `FeatureSet` from a file:

```
from skll.data import Reader

example_reader = Reader.for_path('myexamples.csv')
train_examples = example_reader.read()
```

Or, work with an existing `pandas DataFrame`:

```
from skll.data import FeatureSet

# assuming the data labels are in a column called "y"
train_examples = FeatureSet.from_data_frame(my_data_frame,
                                            "A Name for My Data",
                                            labels_column="y")
```

Train a linear svm (using the already loaded `train_examples`):

```
from skll.learner import Learner

learner = Learner('LinearSVC')
learner.train(train_examples)
```

Evaluate a trained model:

```
test_examples = Reader.for_path('test.tsv').read()
conf_matrix, accuracy, prf_dict, model_params, obj_score = learner.
↪evaluate(test_examples)
```

Perform ten-fold cross-validation with a radial SVM:

```
learner = Learner('SVC')
fold_result_list, grid_search_scores = learner.cross-validate(train_
↪examples)
```

`fold_result_list` in this case is a list of the results returned by `learner.evaluate` for each fold, and `grid_search_scores` is the highest objective function value achieved when tuning the model.

Generate predictions from a trained model:

```
predictions = learner.predict(test_examples)
```

1.7.2 config Package

`skll.config.`**`fix_json`**(*json_string*)

Fix incorrectly formatted quotes and capitalized booleans in JSON string.

Parameters

json_string (*str*) – A JSON-style string.

Returns

The normalized JSON string.

Return type

str

`skll.config.`**`load_cv_folds`**(*folds_file*, *ids_to_floats=False*)

Load cross-validation folds from a CSV file.

The CSV file must contain two columns: example ID and fold ID (and a header).

Parameters

- **folds_file** (*sklI.types.PathOrStr*) – The path to a folds file to read.
- **ids_to_floats** (*bool, default=False*) – Whether to convert IDs to floats.

Returns

Dictionary with example IDs as the keys and fold IDs as the values. If *ids_to_floats* is set to *True*, the example IDs are floats but otherwise they are strings.

Return type

skll.types.FoldMapping

Raises

ValueError – If example IDs cannot be converted to floats and *ids_to_floats* is *True*.

skll.config.**locate_file**(*file_path*, *config_dir*)

Locate a file, given a file path and configuration directory.

Parameters

- **file_path** (*skll.types.PathOrStr*) – The file to locate. Path may be absolute or relative.
- **config_dir** (*skll.types.PathOrStr*) – The path to the configuration file directory.

Returns

path_to_check – The normalized absolute path, if it exists.

Return type

str

Raises

FileNotFoundError – If the file does not exist.

1.7.3 data Package

data.featureset Module

Classes related to storing/merging feature sets.

author

Dan Blanchard (dblanchard@ets.org)

author

Nitin Madnani (nmadnani@ets.org)

author
Jeremy Biggs (jbiggs@ets.org)

organization
ETS

`class skll.data.featureset.`**`FeatureSet`**(*name*, *ids*, *labels=None*, *features=None*, *vectorizer=None*)

Bases: `object`

Encapsulate features, labels, and metadata for a given dataset.

Parameters

- **name** (*str*) – The name of this feature set.
- **ids** (*Union[List[str], numpy.ndarray]*) – Example IDs for this set.
- **labels** (*Optional[Union[List[str], numpy.ndarray], default=None*) – Labels for this set.
- **features** (Optional[Union[*skll.types.FeatureDictList*, `numpy.ndarray`]], default=None) – The features for each instance represented as either a list of dictionaries or a numpy array (if `vectorizer` is also specified).
- **vectorizer** (Optional[Union[`sklearn.feature_extraction.DictVectorizer`, `sklearn.feature_extraction.FeatureHasher`], default=None) – Vectorizer which will be used to generate the feature matrix.

Warning: FeatureSets can only be equal if the order of the instances is identical because these are stored as lists/arrays. Since scikit-learn's `DictVectorizer` automatically sorts the underlying feature matrix if it is sparse, we do not do any sorting before checking for equality. This is not a problem because we _always_ use sparse matrices with `DictVectorizer` when creating FeatureSets.

Notes

If ids, labels, and/or features are not None, the number of rows in each array must be equal.

`filter`(*ids=None*, *labels=None*, *features=None*, *inverse=False*)

Remove or keep features and/or examples from the given feature set.

Filtering is done in-place.

Parameters

- **ids** (Optional[List[*sskll.types.IdType*]], default=None) – Examples to keep in the FeatureSet. If `None`, no ID filtering takes place.
- **labels** (Optional[List[*skll.types.LabelType*]], default=None) – Labels that we want to retain examples for. If `None`, no label filtering takes place.
- **features** (*Optional[List[str]], default=None*) – Features to keep in the FeatureSet. To help with filtering string-valued features that were converted to sequences of boolean features when read in, any features in the FeatureSet that contain a = will be split on the first occurrence and the prefix will be checked to see if it is in `features`. If `None`, no feature filtering takes place. Cannot be used if FeatureSet uses a FeatureHasher for vectorization.
- **inverse** (*bool, default=False*) – Instead of keeping features and/or examples in lists, remove them.

Raises

`ValueError` – If attempting to use features to filter a `FeatureSet` that uses a `FeatureHasher` vectorizer.

Return type

None

filtered_iter(*ids=None*, *labels=None*, *features=None*, *inverse=False*)

Retain only the specified features and/or examples from the output.

Parameters

- **ids** (Optional[List[*skll.types.IdType*]], default=None) – Examples to keep in the `FeatureSet`. If `None`, no ID filtering takes place.
- **labels** (Optional[List[*skll.types.LabelType*]], default=None) – Labels that we want to retain examples for. If `None`, no label filtering takes place.
- **features** (*Optional[Collection[str]], default=None*) – Features to keep in the `FeatureSet`. To help with filtering string-valued features that were converted to sequences of boolean features when read in, any features in the `FeatureSet` that contain a = will be split on the first occurrence and the prefix will be checked to see if it is in `features`. If *None*, no feature filtering takes place. Cannot be used if `FeatureSet` uses a FeatureHasher for vectorization.
- **inverse** (*bool, default=False*) – Instead of keeping features and/or examples in lists, remove them.

Returns

A generator that yields 3-tuples containing:

- *sklL.types.IdType* - The ID of the example.
- *skll.types.LabelType* - The label of the example.
- *skll.types.FeatureDict* - The feature dictionary, with feature name as the key and example value as the value.

Return type
skll.types.FeatGenerator

Raises

- **ValueError** – If the vectorizer is not a `DictVectorizer`.
- **ValueError** – If any of the "labels", "features", or "vectorizer" attribute is `None`.

static from_data_frame(*df*, *name*, *labels_column=None*, *vectorizer=None*)

Create a `FeatureSet` instance from a pandas data frame.

Will raise an Exception if pandas is not installed in your environment. The `ids` in the `FeatureSet` will be the index from the given frame.

Parameters

- **df** (*pandas.DataFrame*) – The pandas.DataFrame object to use as a `FeatureSet`.
- **name** (*str*) – The name of the output `FeatureSet` instance.
- **labels_column** (*Optional[str], default=None*) – The name of the column containing the labels (data to predict).
- **vectorizer** (Optional[Union[`sklearn.feature_extraction.DictVectorizer`, `sklearn.feature_extraction.FeatureHasher`]], default=None) – Vectorizer which will be used to generate the feature matrix.

Returns
A `FeatureSet` instance generated from from the given data frame.

Return type
skll.data.featureset.FeatureSet

property has_labels

Check if `FeatureSet` has finite labels.

Returns
has_labels – Whether or not this FeatureSet has any finite labels.

Return type
bool

static split(*fs*, *ids_for_split1*, *ids_for_split2=None*)

Split `FeatureSet` into two new `FeatureSet` instances.

The splitting is done based on the given indices for the two splits.

Parameters

- **fs** (`skll.data.featureset.FeatureSet`) – The `FeatureSet` instance to split.
- **ids_for_split1** (*`List[int]`*) – A list of example indices which will be split out into the first `FeatureSet` instance. Note that the FeatureSet instance will respect the order of the specified indices.
- **ids_for_split2** (*`Optional[List[int]], default=None`*) – An optional list of example indices which will be split out into the second `FeatureSet` instance. Note that the `FeatureSet` instance will respect the order of the specified indices. If this is not specified, then the second `FeatureSet` instance will contain the complement of the first set of indices sorted in ascending order.

Returns

A tuple containing the two featureset instances.

Return type

Tuple[*`skll.data.featureset.FeatureSet`*, *`skll.data.featureset.FeatureSet`*]

`data.readers` Module

class `skll.data.readers.`**`Reader`**(*path_or_list*, *quiet=True*, *ids_to_floats=False*, *label_col='y'*, *id_col='id'*, *class_map=None*, *sparse=True*, *feature_hasher=False*, *num_features=None*, *logger=None*)

Bases: `object`

Load FeatureSets from files on disk.

This is the base class used to create featureset readers for different file types.

Parameters

- **path_or_list** (Union[*`skll.types.PathOrStr`*, List[Dict[str, Any]]]) – Path or a list of example dictionaries.
- **quiet** (*`bool, default=True`*) – Do not print "Loading…" status message to stderr.
- **ids_to_floats** (*`bool, default=False`*) – Convert IDs to float to save memory. Will raise error if we encounter an a non-numeric ID.

- **label_col** (*Optional[str], default='y'*) – Name of the column which contains the class labels for ARFF/CSV/TSV files. If no column with that name exists, or `None` is specified, the data is considered to be unlabelled.
- **id_col** (*str, default='id'*) – Name of the column which contains the instance IDs. If no column with that name exists, or `None` is specified, example IDs will be automatically generated.
- **class_map** (Optional[*skll.types.ClassMap*], default=None) – Mapping from original class labels to new ones. This is mainly used for collapsing multiple labels into a single class. Anything not in the mapping will be kept the same. The keys are the new labels and the list of values for each key is the labels to be collapsed to said new label.
- **sparse** (*bool, default=True*) – Whether or not to store the features in a numpy CSR matrix when using a DictVectorizer to vectorize the features.
- **feature_hasher** (*bool, default=False*) – Whether or not a FeatureHasher should be used to vectorize the features.
- **num_features** (*Optional[int], default=None*) – If using a FeatureHasher, how many features should the resulting matrix have? You should set this to a power of 2 greater than the actual number of features to avoid collisions.
- **logger** (*Optional[logging.Logger], default=None*) – A logger instance to use to log messages instead of creating a new one by default.

`classmethod for_path`(*path_or_list*, ***kwargs*)

Instantiate Reader sub-class based on the file extension.

If the input is a list of dictionaries instead of a path, use a dictionary reader instead.

Parameters

- **path_or_list** (Union[*skll.types.PathOrStr*, *skll.types.FeatureDictList*]) – A path or list of example dictionaries.
- **kwargs** (*Optional[Dict[str, Any]]*) – The arguments to the Reader object being instantiated.

Returns

reader – A new instance of the Reader sub-class that is appropriate for the given path.

Return type

skll.data.readers.Reader

Raises

`ValueError` – If file does not have a valid extension.

read()

Load examples from various file formats.

The following formats are supported: `.arff`, `.csv`, `.jsonlines`, `.libsvm`, `.ndj`, or `.tsv` formats.

Returns

A `FeatureSet` instance representing the input file.

Return type

sklI.data.featureset.FeatureSet

Raises

- `ValueError` – If `ids_to_floats` is True, but IDs cannot be converted.
- `ValueError` – If no features are found.
- `ValueError` – If the example IDs are not unique.

class `sklI.data.readers.`**CSVReader**(*path_or_list*, *replace_blanks_with=None*, *drop_blanks=False*, *pandas_kwargs=None*, ***kwargs*)

Bases: *Reader*

Create a `FeatureSet` instance from a CSV file.

If example/instance IDs are included in the files, they must be specified in the `id` column. Also, there must be a column with the name specified by `label_col` if the data is labeled.

Parameters

- **path_or_list** (Union[*sklI.types.PathOrStr*, List[Dict[str, Any]]]) – The path to a comma-delimited file.
- **replace_blanks_with** (*Optional[Union[Number, Dict[str, Number]]], default=None*) – Specifies a new value with which to replace blank values. Options are:
  - `Number` : A (numeric) value with which to replace blank values.
  - `dict` : A dictionary specifying the replacement value for each column.
  - `None` : Blank values will be left as blanks, and not replaced.

  The replacement occurs after the data set is read into a `pd.DataFrame`.
- **drop_blanks** (*bool, default=False*) – If `True`, remove lines/rows that have any blank values. These lines/rows are removed after the the data set is read into a `pd.DataFrame`.

- **pandas_kwargs** (*Optional[Dict[str, Any]], default=None*) – Arguments that will be passed directly to the `pandas` I/O reader.
- **kwargs** (*Optional[Dict[str, Any]]*) – Other arguments to the Reader object.

`class sklearn.data.readers.`**`TSVReader`**(*path_or_list*, *replace_blanks_with=None*, *drop_blanks=False*, *pandas_kwargs=None*, ***kwargs*)

Bases: *CSVReader*

Create a `FeatureSet` instance from a TSV file.

If example/instance IDs are included in the files, they must be specified in the `id` column. Also there must be a column with the name specified by `label_col` if the data is labeled.

Parameters

- **path_or_list** (*str*) – The path to a comma-delimited file.
- **replace_blanks_with** (*Optional[Union[Number, Dict[str, Number]]], default=None*) – Specifies a new value with which to replace blank values. Options are:
  - `Number` : A (numeric) value with which to replace blank values.
  - `dict` : A dictionary specifying the replacement value for each column.
  - `None` : Blank values will be left as blanks, and not replaced.

  The replacement occurs after the data set is read into a `pd.DataFrame`.
- **drop_blanks** (*bool, default=False*) – If `True`, remove lines/rows that have any blank values. These lines/rows are removed after the the data set is read into a `pd.DataFrame`.
- **pandas_kwargs** (*Optional[Dict[str, Any]], default=None*) – Arguments that will be passed directly to the `pandas` I/O reader.
- **kwargs** (*Optional[Dict[str, Any]]*) – Other arguments to the Reader object.

`class sklearn.data.readers.`**`NDJReader`**(*path_or_list*, *quiet=True*, *ids_to_floats=False*, *label_col='y'*, *id_col='id'*, *class_map=None*, *sparse=True*, *feature_hasher=False*, *num_features=None*, *logger=None*)

Bases: *Reader*

Create a `FeatureSet` instance from a JSONlines/NDJ file.

If example/instance IDs are included in the files, they must be specified as the “id” key in each JSON dictionary.

Parameters

- **path_or_list** (Union[*skll.types.PathOrStr*, List[Dict[str, Any]]) – Path or a list of example dictionaries.
- **quiet** (*bool, default=True*) – Do not print "Loading…" status message to stderr.
- **ids_to_floats** (*bool, default=False*) – Convert IDs to float to save memory. Will raise error if we encounter an a non-numeric ID.
- **label_col** (*Optional[str], default='y'*) – Name of the column which contains the class labels for ARFF/CSV/TSV files. If no column with that name exists, or `None` is specified, the data is considered to be unlabelled.
- **id_col** (*str, default='id'*) – Name of the column which contains the instance IDs. If no column with that name exists, or `None` is specified, example IDs will be automatically generated.
- **class_map** (Optional[*skll.types.ClassMap*], default=None) – Mapping from original class labels to new ones. This is mainly used for collapsing multiple labels into a single class. Anything not in the mapping will be kept the same. The keys are the new labels and the list of values for each key is the labels to be collapsed to said new label.
- **sparse** (*bool, default=True*) – Whether or not to store the features in a numpy CSR matrix when using a DictVectorizer to vectorize the features.
- **feature_hasher** (*bool, default=False*) – Whether or not a FeatureHasher should be used to vectorize the features.
- **num_features** (*Optional[int], default=None*) – If using a FeatureHasher, how many features should the resulting matrix have? You should set this to a power of 2 greater than the actual number of features to avoid collisions.
- **logger** (*Optional[logging.Logger], default=None*) – A logger instance to use to log messages instead of creating a new one by default.

class `skll.data.readers.`**DictListReader**(*path_or_list*, *quiet=True*, *ids_to_floats=False*, *label_col='y'*, *id_col='id'*, *class_map=None*, *sparse=True*, *feature_hasher=False*, *num_features=None*, *logger=None*)

Bases: *Reader*

Facilitate programmatic use of methods that take `FeatureSet` as input.

Support `Learner.predict()` and other methods that take `FeatureSet` objects as input. It iterates over examples in the same way as other `Reader` classes, but uses a list of example dictionaries instead of a path to a file.

Parameters

- **`path_or_list`** (Union[*`skll.types.PathOrStr`*, List[Dict[str, Any]]) – Path or a list of example dictionaries.
- **`quiet`** (*`bool, default=True`*) – Do not print "Loading…" status message to stderr.
- **`ids_to_floats`** (*`bool, default=False`*) – Convert IDs to float to save memory. Will raise error if we encounter an a non-numeric ID.
- **`label_col`** (*`Optional[str], default='y'`*) – Name of the column which contains the class labels for ARFF/CSV/TSV files. If no column with that name exists, or `None` is specified, the data is considered to be unlabelled.
- **`id_col`** (*`str, default='id'`*) – Name of the column which contains the instance IDs. If no column with that name exists, or `None` is specified, example IDs will be automatically generated.
- **`class_map`** (Optional[*`skll.types.ClassMap`*], default=None) – Mapping from original class labels to new ones. This is mainly used for collapsing multiple labels into a single class. Anything not in the mapping will be kept the same. The keys are the new labels and the list of values for each key is the labels to be collapsed to said new label.
- **`sparse`** (*`bool, default=True`*) – Whether or not to store the features in a numpy CSR matrix when using a DictVectorizer to vectorize the features.
- **`feature_hasher`** (*`bool, default=False`*) – Whether or not a FeatureHasher should be used to vectorize the features.
- **`num_features`** (*`Optional[int], default=None`*) – If using a FeatureHasher, how many features should the resulting matrix have? You should set this to a power of 2 greater than the actual number of features to avoid collisions.
- **`logger`** (*`Optional[logging.Logger], default=None`*) – A logger instance to use to log messages instead of creating a new one by default.

`read()`

Read examples from list of dictionaries.

Returns

A `FeatureSet` representing the list of dictionaries we read in.

Return type

`skll.data.FeatureSet`

class sklearn.data.readers.**ARFFReader**(*path_or_list*, ***kwargs*)

Bases: `Reader`

Create a `FeatureSet` instance from an ARFF file.

If example/instance IDs are included in the files, they must be specified in the `id` column. Also, there must be a column with the name specified by `label_col` if the data is labeled, and this column must be the final one (as it is in Weka).

Parameters

- **path_or_list** (Union[*skll.types.PathOrStr*, List[Dict[str, Any]]]) – The path to the ARFF file.
- **kwargs** (*Optional[Dict[str, Any]]*) – Other arguments to the Reader object.

static split_with_quotes(*string*, *delimiter=' '*, *quote_char="'"*, *escape_char='\\'*)

Split strings but not on split delimiters enclosed in quotes.

Parameters

- **string** (*str*) – The string with quotes to split
- **delimiter** (*str, default=' '*) – The delimiter to split on.
- **quote_char** (*str, default="'"*) – The quote character to ignore.
- **escape_char** (*str, default='\'*) – The escape character.

Return type

List[str]

class sklearn.data.readers.**LibSVMReader**(*path_or_list*, *quiet=True*, *ids_to_floats=False*, *label_col='y'*, *id_col='id'*, *class_map=None*, *sparse=True*, *feature_hasher=False*, *num_features=None*, *logger=None*)

Bases: `Reader`

Create a `FeatureSet` instance from a LibSVM/LibLinear/SVMLight file.

We use a specially formatted comment for storing example IDs, class names, and feature names, which are normally not supported by the format. The comment is not mandatory, but without it, your labels and features will not have names. The comment is structured as follows:

```
ExampleID | 1=FirstClass | 1=FirstFeature 2=SecondFeature
```

Parameters

- **path_or_list** (Union[*skll.types.PathOrStr*, List[Dict[str, Any]]]) – Path or a list of example dictionaries.

- **quiet** (*bool, default=True*) – Do not print "Loading…" status message to stderr.
- **ids_to_floats** (*bool, default=False*) – Convert IDs to float to save memory. Will raise error if we encounter an a non-numeric ID.
- **label_col** (*Optional[str], default='y'*) – Name of the column which contains the class labels for ARFF/CSV/TSV files. If no column with that name exists, or `None` is specified, the data is considered to be unlabelled.
- **id_col** (*str, default='id'*) – Name of the column which contains the instance IDs. If no column with that name exists, or `None` is specified, example IDs will be automatically generated.
- **class_map** (Optional[*skll.types.ClassMap*], default=None) – Mapping from original class labels to new ones. This is mainly used for collapsing multiple labels into a single class. Anything not in the mapping will be kept the same. The keys are the new labels and the list of values for each key is the labels to be collapsed to said new label.
- **sparse** (*bool, default=True*) – Whether or not to store the features in a numpy CSR matrix when using a DictVectorizer to vectorize the features.
- **feature_hasher** (*bool, default=False*) – Whether or not a FeatureHasher should be used to vectorize the features.
- **num_features** (*Optional[int], default=None*) – If using a FeatureHasher, how many features should the resulting matrix have? You should set this to a power of 2 greater than the actual number of features to avoid collisions.
- **logger** (*Optional[logging.Logger], default=None*) – A logger instance to use to log messages instead of creating a new one by default.

`data.writers` Module

`class skll.data.writers.`**`Writer`**(*path*, *feature_set*, *quiet=True*, *subsets=None*, *logger=None*)

Bases: `object`

Write out FeatureSets to files on disk.

This is the base class used to create featureset writers for different file types.

Parameters

- **path** (*skll.types.PathOrStr*) – A path to the feature file we would like to create. The suffix to this filename must be `.arff`, `.csv`, `.jsonlines`, `.libsvm`, `.ndj`, or `.tsv`. If `subsets` is not `None`, when calling the `write()` method, path is assumed to be a string containing the path to the directory to write the feature files with an additional file extension specifying the file type. For example `/foo/.csv`.
- **feature_set** (*skll.data.featureset.FeatureSet*) – The `FeatureSet` instance to dump to the file.
- **quiet** (*bool, default=True*) – Do not print "Writing…" status message to stderr.
- **subsets** (*Optional[Dict[str, List[str]]], default=None*) – A mapping from subset names to lists of feature names that are included in those sets. If given, a feature file will be written for every subset (with the name containing the subset name as suffix to `path`). Note, since string-valued features are automatically converted into boolean features with names of the form `FEATURE_NAME=STRING_VALUE`, when doing the filtering, the portion before the = is all that's used for matching. Therefore, you do not need to enumerate all of these boolean feature names in your mapping.
- **logger** (*Optional[logging.Logger], default=None*) – A logger instance to use to log messages instead of creating a new one by default.

classmethod for_path(*path*, *feature_set*, ***kwargs*)

Retrieve object of `Writer` sub-class appropriate for given path.

Parameters

- **path** (*skll.types.PathOrStr*) – A path to the feature file we would like to create. The suffix to this filename must be `.arff`, `.csv`, `.jsonlines`, `.libsvm`, `.ndj`, or `.tsv`. If `subsets` is not `None`, when calling the `write()` method, path is assumed to be a string containing the path to the directory to write the feature files with an additional file extension specifying the file type. For example `/foo/.csv`.
- **feature_set** (*skll.data.featureset.FeatureSet*) – The `FeatureSet` instance to dump to the output file.
- **kwargs** (*Optional[Dict[str, Any]]*) – The keyword arguments for `for_path` are the same as the initializer for the desired `Writer` subclass.

Returns

writer – New instance of the Writer sub-class that is appropriate for the given path.

Return type
`skll.data.Writer`

write()

Write out this Writer's `FeatureSet` to a file in its format.

Return type
None

class `skll.data.writers.`**CSVWriter**(*path*, *feature_set*, *quiet=True*, *subsets=None*, *logger=None*, *label_col='y'*, *id_col='id'*, *pandas_kwargs=None*)

Bases: *Writer*

Writer for writing out `FeatureSet` instances as CSV files.

Parameters

- **path** (*skll.types.PathOrStr*) – A path to the feature file we would like to create. If `subsets` is not `None`, this is assumed to be a string containing the path to the directory to write the feature files with an additional file extension specifying the file type. For example `/foo/.csv`.
- **feature_set** (*skll.data.featureset.FeatureSet*) – The `FeatureSet` instance to dump to the output file.
- **quiet** (*bool, default=True*) – Do not print "Writing..." status message to stderr.
- **subsets** (*Optional[Dict[str, List[str]]], default=None*) – A mapping from subset names to lists of feature names that are included in those sets. If given, a feature file will be written for every subset (with the name containing the subset name as suffix to `path`). Note, since string-valued features are automatically converted into boolean features with names of the form `FEATURE_NAME=STRING_VALUE`, when doing the filtering, the portion before the = is all that's used for matching. Therefore, you do not need to enumerate all of these boolean feature names in your mapping.
- **logger** (*Optional[logging.Logger], default=None*) – A logger instance to use to log messages instead of creating a new one by default.
- **label_col** (*str, default="y"*) – The column name containing the label.
- **id_col** (*str, default="id"*) – The column name containing the ID.
- **pandas_kwargs** (*Optional[Dict[str], Any], default=None*) – Arguments that will be passed directly to the *pandas* I/O reader.

class `skll.data.writers.`**`TSVWriter`**(*path*, *feature_set*, *quiet=True*, *subsets=None*, *logger=None*, *label_col='y'*, *id_col='id'*, *pandas_kwargs=None*)

Bases: *`CSVWriter`*

Writer for writing out FeatureSets as TSV files.

Parameters

- **`path`** (*`skll.types.PathOrStr`*) – A path to the feature file we would like to create. If `subsets` is not `None`, this is assumed to be a string containing the path to the directory to write the feature files with an additional file extension specifying the file type. For example `/foo/.tsv`.
- **`feature_set`** (*`skll.data.featureset.FeatureSet`*) – The `FeatureSet` instance to dump to the output file.
- **`quiet`** (*`bool, default=True`*) – Do not print "Writing…" status message to stderr.
- **`subsets`** (*`Optional[Dict[str, List[str]]], default=None`*) – A mapping from subset names to lists of feature names that are included in those sets. If given, a feature file will be written for every subset (with the name containing the subset name as suffix to `path`). Note, since string-valued features are automatically converted into boolean features with names of the form `FEATURE_NAME=STRING_VALUE`, when doing the filtering, the portion before the = is all that's used for matching. Therefore, you do not need to enumerate all of these boolean feature names in your mapping.
- **`logger`** (*`Optional[logging.Logger], default=None`*) – A logger instance to use to log messages instead of creating a new one by default.
- **`label_col`** (*`str, default="y"`*) – The column name containing the label.
- **`id_col`** (*`str, default="id"`*) – The column name containing the ID.
- **`pandas_kwargs`** (*`Optional[Dict[str, Any]], default=None`*) – Arguments that will be passed directly to the *pandas* I/O reader.

class `skll.data.writers.`**`NDJWriter`**(*path*, *feature_set*, *quiet=True*, *subsets=None*, *logger=None*)

Bases: *`Writer`*

Writer for writing out FeatureSets as .jsonlines/.ndj files.

Parameters

- **`path`** (*`skll.types.PathOrStr`*) – A path to the feature file we would like to create. If `subsets` is not `None`, this is assumed to be a string containing the path to the directory to write the feature files with an additional file extension specifying the file type. For example `/foo/.ndj`.

- **feature_set** (*skll.data.featureset.FeatureSet*) – The `FeatureSet` instance to dump to the output file.
- **quiet** (*bool, default=True*) – Do not print "Writing..." status message to stderr.
- **subsets** (*Optional[Dict[str, List[str]]], default=None*) – A mapping from subset names to lists of feature names that are included in those sets. If given, a feature file will be written for every subset (with the name containing the subset name as suffix to `path`). Note, since string-valued features are automatically converted into boolean features with names of the form `FEATURE_NAME=STRING_VALUE`, when doing the filtering, the portion before the = is all that's used for matching. Therefore, you do not need to enumerate all of these boolean feature names in your mapping.
- **logger** (*Optional[logging.Logger], default=None*) – A logger instance to use to log messages instead of creating a new one by default.

class `skll.data.writers.`**ARFFWriter**(*path*, *feature_set*, *quiet=True*, *subsets=None*, *logger=None*, *relation='skll_relation'*, *regression=False*, *dialect='excel-tab'*, *label_col='y'*, *id_col='id'*)

Bases: *Writer*

Writer for writing out FeatureSets as ARFF files.

Parameters

- **path** (*skll.types.PathOrStr*) – A path to the feature file we would like to create. If `subsets` is not `None`, this is assumed to be a string containing the path to the directory to write the feature files with an additional file extension specifying the file type. For example `/foo/.arff`.
- **feature_set** (*skll.data.featureset.FeatureSet*) – The `FeatureSet` instance to dump to the output file.
- **quiet** (*bool, default=True*) – Do not print "Writing..." status message to stderr.
- **subsets** (*Optional[Dict[str, List[str]]], default=None*) – A mapping from subset names to lists of feature names that are included in those sets. If given, a feature file will be written for every subset (with the name containing the subset name as suffix to `path`). Note, since string-valued features are automatically converted into boolean features with names of the form `FEATURE_NAME=STRING_VALUE`, when doing the filtering, the portion before the = is all that's used for matching. Therefore,

you do not need to enumerate all of these boolean feature names in your mapping.

- **logger** (*Optional[logging.Logger], default=None*) – A logger instance to use to log messages instead of creating a new one by default.
- **relation** (*str, default='skll_relation'*) – The name of the relation in the ARFF file.
- **regression** (*bool, default=False*) – Is this an ARFF file to be used for regression?
- **kwargs** (*Optional[Dict[str, Any]]*) – The arguments to the `Writer` object being instantiated.

class sll.data.writers.**LibSVMWriter**(*path*, *feature_set*, *quiet=True*, *subsets=None*, *logger=None*, *label_map=None*)

Bases: *Writer*

Writer for writing out FeatureSets as LibSVM/SVMLight files.

Parameters

- **path** (*skll.types.PathOrStr*) – A path to the feature file we would like to create. If `subsets` is not `None`, this is assumed to be a string containing the path to the directory to write the feature files with an additional file extension specifying the file type. For example `/foo/.libsvm`.
- **feature_set** (*skll.data.featureset.FeatureSet*) – The `FeatureSet` instance to dump to the output file.
- **quiet** (*bool, default=True*) – Do not print "Writing…" status message to stderr.
- **subsets** (*Optional[Dict[str, List[str]]], default=None*) – A mapping from subset names to lists of feature names that are included in those sets. If given, a feature file will be written for every subset (with the name containing the subset name as suffix to `path`). Note, since string-valued features are automatically converted into boolean features with names of the form `FEATURE_NAME=STRING_VALUE`, when doing the filtering, the portion before the = is all that's used for matching. Therefore, you do not need to enumerate all of these boolean feature names in your mapping.
- **logger** (*Optional[logging.Logger], default=None*) – A logger instance to use to log messages instead of creating a new one by default.
- **label_map** (*Optional[Dict[str, int]], default=None*) – A mapping from label strings to integers.

1.7.4 `experiments` Package

1.7.5 `learner` Package

`Learner` Class

An easy-to-use class that wraps scikit-learn estimators.

author
Nitin Madnani (nmadnani@ets.org)

author
Michael Heilman (mheilman@ets.org)

author
Dan Blanchard (dblanchard@ets.org)

author
Aoife Cahill (acahill@ets.org)

organization
ETS

class `skll.learner.`**`Learner`**(*model_type*, *probability=False*, *pipeline=False*, *feature_scaling='none'*, *model_kwargs=None*, *pos_label=None*, *min_feature_count=1*, *sampler=None*, *sampler_kwargs=None*, *custom_learner_path=None*, *logger=None*)

Bases: `object`

A simpler interface around scikit-learn classification and regression estimators.

Parameters

- **`model_type`** (*`str`*) – Name of estimator to create (e.g., `'LogisticRegression'`). See the skll package documentation for valid options.
- **`probability`** (*`bool`*, *`default=False`*) – Should learner return probabilities of all labels (instead of just label with highest probability)?
- **`pipeline`** (*`bool`*, *`default=False`*) – Should learner contain a pipeline attribute that contains a scikit-learn Pipeline object composed of all steps including the vectorizer, the feature selector, the sampler, the feature scaler, and the actual estimator. Note that this will increase the size of the learner object in memory and also when it is saved to disk.
- **`feature_scaling`** (*`str`*, *`default="none"`*) – How to scale the features, if at all. Options are - 'with_std': scale features using the standard

deviation - 'with_mean': center features using the mean - 'both': do both scaling as well as centering - 'none': do neither scaling nor centering

- **model_kwargs** (*Optional[Dict[str, Any]], default=None*) – A dictionary of keyword arguments to pass to the initializer for the specified model.
- **pos_label** (Optional[*skll.types.LabelType*], default=None) – An integer or string denoting the label of the class to be treated as the positive class in a binary classification setting. If `None`, the class represented by the label that appears second when sorted is chosen as the positive class. For example, if the two labels in data are "A" and "B" and `pos_label` is not specified, "B" will be chosen as the positive class.
- **min_feature_count** (*int, default=1*) – The minimum number of examples a feature must have a nonzero value in to be included.
- **sampler** (*Optional[str], default=None*) – The sampler to use for kernel approximation, if desired. Valid values are - 'AdditiveChi2Sampler' - 'Nystroem' - 'RBFSampler' - 'SkewedChi2Sampler'
- **sampler_kwargs** (*Optional[Dict[str, Any]], default=None*) – A dictionary of keyword arguments to pass to the initializer for the specified sampler.
- **custom_learner_path** (*Optional[str], default=None*) – Path to module where a custom classifier is defined.
- **logger** (*Optional[logging.Logger], default=None*) – A logging object. If `None` is passed, get logger from `__name__`.

cross_validate(*examples*, *stratified=True*, *cv_folds=10*, *cv_seed=123456789*, *grid_search=True*, *grid_search_folds=5*, *grid_jobs=None*, *grid_objective=None*, *output_metrics=[]*, *prediction_prefix=None*, *param_grid=None*, *shuffle=False*, *save_cv_folds=True*, *save_cv_models=False*, *use_custom_folds_for_grid_search=True*)

Cross-validate the learner on the given training examples.

Parameters

- **examples** (*sklll.data.featureset.FeatureSet*) – The `FeatureSet` instance to cross-validate learner performance on.
- **stratified** (*bool, default=True*) – Should we stratify the folds to ensure an even distribution of labels for each fold?
- **cv_folds** (Union[int, *skll.types.FoldMapping*], default=10) – The number of folds to use for cross-validation, or a mapping from example IDs to folds.

- **cv_seed** (*int, default=123456789*) – The value for seeding the random number generator used to create the random folds. Note that this seed is *only* used if either `grid_search` or `shuffle` are set to `True`.
- **grid_search** (*bool, default=True*) – Should we do grid search when training each fold? Note: This will make this take *much* longer.
- **grid_search_folds** (Union[int, *skll.types.FoldMapping*], default=5) – The number of folds to use when doing the grid search, or a mapping from example IDs to folds.
- **grid_jobs** (*Optional[int], default=None*) – The number of jobs to run in parallel when doing the grid search. If `None` or 0, the number of grid search folds will be used.
- **grid_objective** (*Optional[str], default=None*) – The name of the objective function to use when doing the grid search. Must be specified if `grid_search` is `True`.
- **output_metrics** (*List[str], default = []*) – List of additional metric names to compute in addition to the metric used for grid search.
- **prediction_prefix** (*Optional[str], default=None*) – If saving the predictions, this is the prefix that will be used for the filename. It will be followed by `"_predictions.tsv"`
- **param_grid** (*Optional[Dict[str, Any]], default=None*) – The parameter grid to search.
- **shuffle** (*bool, default=False*) – Shuffle examples before splitting into folds for CV.
- **save_cv_folds** (*bool, default=True*) – Whether to save the cv fold ids or not?
- **save_cv_models** (*bool, default=False*) – Whether to save the cv models or not?
- **use_custom_folds_for_grid_search** (*bool, default=True*) – If `cv_folds` is a custom dictionary, but `grid_search_folds` is not, perhaps due to user oversight, should the same custom dictionary automatically be used for the inner grid-search cross-validation?

Returns

A 5-tuple containing the following:

List[*skll.types.EvaluateTaskResults*]: the confusion matrix, overall accuracy, per-label PRFs, model parameters, objective function score, and evaluation metrics (if any) for each fold.

List[float]: the grid search scores for each fold.

List[Dict[str, Any]]: list of dictionaries of grid search CV results, one per fold, with keys such as "params", "mean_test_score", etc, that are mapped to lists of values associated with each hyperparameter set combination.

Optional[*skll.types.FoldMapping*]: dictionary containing the test-fold number for each id if `save_cv_folds` is `True`, otherwise `None`.

Optional[List[*skll.learner.Learner*]]: list of learners, one for each fold if `save_cv_models` is `True`, otherwise `None`.

Return type
skll.types.CrossValidateTaskResults

Raises

- `ValueError` – If classification labels are not properly encoded as strings.
- `ValueError` – If `grid_search` is `True` but `grid_objective` is `None`.

evaluate(*examples*, *prediction_prefix=None*, *append=False*, *grid_objective=None*, *output_metrics=[]*)

Evaluate the learner on a given dev or test `FeatureSet`.

Parameters

- **examples** (*skll.data.featureset.FeatureSet*) – The `FeatureSet` instance to evaluate the performance of the model on.
- **prediction_prefix** (*Optional[str], default=None*) – If not `None`, predictions will also be written out to a file with the name `<prediction_prefix>_predictions.tsv`. Note that the prefix can also contain a path.
- **append** (*bool, default=False*) – Should we append the current predictions to the file if it exists?
- **grid_objective** (*Optional[str], default=None*) – The objective function that was used when doing the grid search.
- **output_metrics** (*List[str], default=[]*) – List of additional metric names to compute in addition to grid objective.

Returns
A 6-tuple containing the confusion matrix, the overall accuracy, the per-label PRFs, the model parameters, the grid search objective function score,

and the additional evaluation metrics, if any. For regressors, the first two elements in the tuple are `None`.

Return type
`sklearn.types.EvaluateTaskResults`

classmethod from_file(*learner_path*, *logger=None*)

Load a saved `Learner` instance from a file path.

Parameters

- **learner_path** (*`sklearn.types.PathOrStr`*) – The path to a saved `Learner` instance file.
- **logger** (*`Optional[logging.Logger], default=None`*) – A logging object. If `None` is passed, get logger from `__name__`.

Returns
The `Learner` instance loaded from the file.

Return type
`sklearn.learner.Learner`

get_feature_names_out()

Return the names of the actual features used by the estimator.

It is possible for some features to get filtered out by the feature selector which means that the vectorizer is no longer the correct source for the feature names. This method takes into account the feature selector and returns the names of the features that were actually selected to be used by the estimator.

Returns
names – Names of features actually used by the estimator.

Return type
numpy.ndarray of shape (num_features,)

Raises
`ValueError` – If `self.feat_vectorizer` is either `None` or a `sklearn.feature_extraction.FeatureHasher`.

learning_curve(*examples*, *metric*, *cv_folds=10*, *train_sizes=array([0.1, 0.325, 0.55, 0.775, 1.])*, *override_minimum=False*)

Generate learning curves for the learner using the examples.

The learning curves are generated on the training examples via cross-validation. Adapted from the scikit-learn code for learning curve generation (cf.``sklearn.model_selection.learning_curve``).

Parameters

- **examples** (*sk11.data.featureset.FeatureSet*) – The FeatureSet instance to generate the learning curve on.
- **cv_folds** (Union[int, *skll.types.FoldMapping*], default=10) – The number of folds to use for cross-validation, or a mapping from example IDs to folds.
- **metric** (*str*) – The name of the metric function to use when computing the train and test scores for the learning curve.
- **train_sizes** (*skll.types.LearningCurveSizes*, default= numpy.linspace() with start=0.1, stop=1.0, num=5) – Relative or absolute numbers of training examples that will be used to generate the learning curve. If the type is float, it is regarded as a fraction of the maximum size of the training set (that is determined by the selected validation method), i.e. it has to be within (0, 1]. Otherwise it is interpreted as absolute sizes of the training sets. Note that for classification the number of samples usually have to be big enough to contain at least one sample from each class.
- **override_minimum** (*bool, default=False*) – Learning curves can be unreliable for very small sizes esp. for > 2 labels. If this option is set to True, the learning curve would be generated even if the number of example is less 500 along with a warning. If False, the curve is not generated and an exception is raised instead.

Returns

- **train_scores** (*List[float]*) – The scores for the training set.
- **test_scores** (*List[float]*) – The scores on the test set.
- **fit_times** (*List[float]*) – The average times taken to fit each model.
- **num_examples** (*List[int]*) – The numbers of training examples used to generate the curve.

Raises

ValueError – If the number of examples is less than 500.

Return type

Tuple[*List*[float], *List*[float], *List*[float], *List*[int]]

load(*learner_path*)

Replace the current learner instance with a saved learner.

Parameters

learner_path (*skll.types.PathOrStr*) – The path to a saved learner object file to load.

Return type
None

property model

Return the underlying scikit-learn model.

property model_kwargs: Dict[str, Any]

Return a dictionary of the underlying scikit-learn model's keyword arguments.

property model_params: Tuple[Dict[str, Any], Dict[str, Any]]

Return model parameters (i.e., weights).

Return the weights for a `LinearModel` (e.g., `Ridge`), regression, and liblinear models. If the model was trained using feature hashing, then names of the form *hashed_feature_XX* are used instead.

Returns

- **res** (*Dict[str, Any]*) – A dictionary of labeled weights.
- **intercept** (*Dict[str, Any]*) – A dictionary of intercept(s).

Raises
`ValueError` – If the instance does not support model parameters.

property model_type

Return the model type (i.e., the class).

predict(*examples*, *prediction_prefix=None*, *append=False*, *class_labels=True*)

Generate predictions for the given examples using the learner model.

Return, and optionally, write out predictions on a given `FeatureSet` to a file. For regressors, the returned and written-out predictions are identical. However, for classifiers:

- if `class_labels` is `True`, class labels are returned as well as written out.
- if `class_labels` is `False` and the classifier is probabilistic (i.e., `self..probability` is `True`), class probabilities are returned as well as written out.
- if `class_labels` is `False` and the classifier is non-probabilistic (i.e., `self..probability` is `False`), class indices are returned and class labels are written out.

TL;DR: for regressors, just ignore `class_labels`. For classfiers, set it to `True` to get class labels and `False` to get class probabilities.

Parameters

- **examples** (*skll.data.featureset.FeatureSet*) – The `FeatureSet` instance to predict labels for.
- **prediction_prefix** (*Optional[str], default=None*) – If not `None`, predictions will also be written out to a file with the name

`<prediction_prefix>_predictions.tsv`. For classifiers, the predictions written out are class labels unless the learner is probabilistic AND `class_labels` is set to `False`. Note that this prefix can also contain a path.

- **append** (*bool, default=False*) – Should we append the current predictions to the file if it exists?
- **class_labels** (*bool, default=True*) – If `False`, return either the class probabilities (probabilistic classifiers) or the class indices (non-probabilistic ones). If `True`, return the class labels no matter what. Ignored for regressors.

Returns
The predictions returned by the `Learner` instance.

Return type
numpy.ndarray

Raises

- **AssertionError** – If invalid predictions are being returned or written out.
- **MemoryError** – If process runs out of memory when converting to dense.
- **RuntimeError** – If there is a mismatch between the learner vectorizer and the test set vectorizer.

property probability: bool

Return the value of the probability flag.

The flag indicages whether the learner return probabilities of all labels (instead of just label with highest probability)?

save(*learner_path*)

Save the `Learner` instance to a file.

Parameters
learner_path (*skll.types.PathOrStr*) – The path to save the `Learner` instance to.

Return type
None

train(*examples, param_grid=None, grid_search_folds=5, grid_search=True, grid_objective=None, grid_jobs=None, shuffle=False*)

Train model underlying the learner.

Return the grid search score and a dictionary of grid search results.

Parameters

- **examples** (*skll.data.featureset.FeatureSet*) – The `FeatureSet` instance to use for training.
- **param_grid** (*Optional[Dict[str, Any]], default=None*) – The parameter grid to search through for grid search. If `None`, a default parameter grid will be used.
- **grid_search_folds** (Union[int, *skll.types.FoldMapping*], default=5) – The number of folds to use when doing the grid search, or a mapping from example IDs to folds.
- **grid_search** (*bool, default=True*) – Should we do grid search?
- **grid_objective** (*Optional[str], default=None*) – The name of the objective function to use when doing the grid search. Must be specified if `grid_search` is `True`.
- **grid_jobs** (*Optional[int], default=None*) – The number of jobs to run in parallel when doing the grid search. If `None` or 0, the number of grid search folds will be used.
- **shuffle** (*bool, default=False*) – Shuffle examples (e.g., for grid search CV.)

Returns

- *float* – The best grid search objective function score, or 0 if we're not doing grid search
- *Dict[str, Any]* – Dictionary of grid search CV results with keys such as "params", "mean_test_score", etc, that are mapped to lists of values associated with each hyperparameter set combination, or None if not doing grid search.

Raises

- **ValueError** – If grid_objective is not a valid grid objective or if one is not specified when necessary.
- **MemoryError** – If process runs out of memory converting training data to dense.
- **ValueError** – If FeatureHasher is used with MultinomialNB.

Return type

Tuple[float, *Dict*[str, *Any*]]

`skll.learner.`**`load_custom_learner`**(*custom_learner_path*, *custom_learner_name*)

Import and load the custom learner object from the given path.

Parameters

- **custom_learner_path** (*sklll.types.PathOrStr*) – The path to a custom learner.
- **custom_learner_name** (*str*) – The name of a custom learner.

Returns
The SKLL learner object loaded from the given path.

Return type
skll.learner.Learner

Raises
ValueError – If the custom learner path does not end in '.py'.

`VotingLearner` Class

A meta-learner class that wraps scikit-learn's *VotingClassifier* and *VotingRegressor*.

author
Nitin Madnani (nmadnani@ets.org)

organization
ETS

class `skll.learner.voting.`**`VotingLearner`**(*learner_names*, *voting='hard'*, *custom_learner_path=None*, *feature_scaling='none'*, *pos_label=None*, *min_feature_count=1*, *model_kwargs_list=None*, *sampler_list=None*, *sampler_kwargs_list=None*, *logger=None*)

Bases: `object`

Wrap `VotingClassifier` and `VotingRegressor` from scikit-learn.

Note that this class does not inherit from the `Learner` class but rather uses different `Learner` instances underlyingly.

Parameters

- **learner_names** (*List[str]*) – List of the learner names that will participate in the voting process.
- **voting** (*Optional[str], default="hard"*) – One of "hard" or "soft". If "hard", the predicted class labels are used for majority rule voting. If "soft", the predicted class label is based on the argmax of the sums of the predicted probabilities from each of the underlying learnrs. This parameter is only relevant for classification.

- **custom_learner_path** (Optional[*skll.types.PathOrStr*], default=None) – Path to a Python file containing the definitions of any custom learners. Any and all custom learners in `estimator_names` must be defined in this file. If the custom learner does not inherit from an already existing scikit-learn estimator, it must explicitly define an *_estimator_type* attribute indicating whether it's a "classifier" or a "regressor".
- **feature_scaling** (*str, default="none"*) – How to scale the features, if at all for each estimator. Options are - "with_std": scale features using the standard deviation - "with_mean": center features using the mean - "both": do both scaling as well as centering - "none": do neither scaling nor centering
- **pos_label** (Optional[*skll.types.LabelType*], default=None) – A string denoting the label of the class to be treated as the positive class in a binary classification setting, for each estimator. If `None`, the class represented by the label that appears second when sorted is chosen as the positive class. For example, if the two labels in data are "A" and "B" and `pos_label` is not specified, "B" will be chosen as the positive class.
- **min_feature_count** (*int, default=1*) – The minimum number of examples a feature must have a nonzero value in to be included, for each estimator.
- **model_kwargs_list** (*Optional[List[Dict[str, Any]]], default=None*) – A list of dictionaries of keyword arguments to pass to the initializer for each of the estimators. There's a one-to-one correspondence between the order of this list and the order of the `learner_names` list.
- **sampler_list** (*Optional[List[str]], default=None*) – The samplers to use for kernel approximation, if desired, for each estimator. Valid values are: - "AdditiveChi2Sampler" - "Nystroem" - "RBFSampler" - "SkewedChi2Sampler" There's a one-to-one correspondence between the order of this list and the order of the `learner_names` list.
- **sampler_kwargs_list** (*Optional[List[Dict[str, Any]]], default=None*) – A list of dictionaries of keyword arguments to pass to the initializer for the specified sampler, one per estimator. There's a one-to-one correspondence between the order of this list and the order of the `learner_names` list.
- **logger** (*Optional[logging.Logger], default=None*) – A logging object. If `None` is passed, get logger from `__name__`.

`cross_validate`(*examples*, *stratified=True*, *cv_folds=10*, *cv_seed=123456789*, *grid_search=True*, *grid_search_folds=5*, *grid_jobs=None*, *grid_objective=None*, *output_metrics=[]*, *prediction_prefix=None*, *param_grid_list=None*, *shuffle=False*, *save_cv_folds=True*, *save_cv_models=False*, *individual_predictions=False*, *use_custom_folds_for_grid_search=True*)

Cross-validate the meta-estimator on the given examples.

We follow essentially the same methodology as in `Learner.cross_validate()` - split the examples into training and testing folds, and then call `self.train()` on the training folds and then `self.evaluate()` on the test fold. Note that this means that underlying estimators with different hyperparameters may be used for each fold, as is the case with `Learner.cross_validate()`.

Parameters

- **`examples`** (*`skll.data.featureset.FeatureSet`*) – The `FeatureSet` instance to cross-validate learner performance on.
- **`stratified`** (*`bool`, `default=True`*) – Should we stratify the folds to ensure an even distribution of labels for each fold?
- **`cv_folds`** (Union[int, *`skll.types.FoldMapping`*], default=10) – The number of folds to use for cross-validation, or a mapping from example IDs to folds.
- **`cv_seed`** (*`int`, `default=123456789`*) – The value for seeding the random number generator used to create the random folds. Note that this seed is *only* used if either `grid_search` or `shuffle` are set to `True`.
- **`grid_search`** (*`bool`, `default=True`*) – Should we do grid search when training each fold? Note: This will make this take *much* longer.
- **`grid_search_folds`** (Union[int, *`skll.types.FoldMapping`*], default=5) – The number of folds to use when doing the grid search, or a mapping from example IDs to folds.
- **`grid_jobs`** (*`Optional[int]`, `default=None`*) – The number of jobs to run in parallel when doing the grid search. If `None` or 0, the number of grid search folds will be used.
- **`grid_objective`** (*`Optional[str]`, `default=None`*) – The name of the objective function to use when doing the grid search. Must be specified if `grid_search` is `True`.
- **`output_metrics`** (*`Optional[List[str]]`, `default=[]`*) – List of additional metric names to compute in addition to the metric used for grid search.

- **prediction_prefix** (*Optional[str], default=None*) – If saving the predictions, this is the prefix that will be used for the filename. It will be followed by `"_predictions.tsv"`
- **param_grid_list** (*Optional[List[Dict[str, Any]]], default=None*) – The list of parameters grid to search through for grid search, one for each underlying learner. The order of the dictionaries should correspond to the order If `None`, the default parameter grids will be used for the underlying estimators.
- **shuffle** (*bool, default=False*) – Shuffle examples before splitting into folds for CV.
- **save_cv_folds** (*bool, default=True*) – Whether to save the cv fold ids or not?
- **save_cv_models** (*bool, default=False*) – Whether to save the cv models or not?
- **individual_predictions** (*bool, default=False*) – Write out the cross-validated predictions from each underlying learner as well.
- **use_custom_folds_for_grid_search** (*bool, default=True*) – If `cv_folds` is a custom dictionary, but `grid_search_folds` is not, perhaps due to user oversight, should the same custom dictionary automatically be used for the inner grid-search cross-validation?

Returns

A 3-tuple containing the following:

List[*skll.types.EvaluateTaskResults*]: the confusion matrix, overall accuracy, per-label PRFs, model parameters, objective function score, and evaluation metrics (if any) for each fold.

Optional[*skll.types.FoldMapping*]: dictionary containing the test-fold number for each id if `save_cv_folds` is `True`, otherwise `None`.

Optional[List[*skll.learner.voting.VotingLearner*]]: list of voting learners, one for each fold if `save_cv_models` is `True`, otherwise `None`.

Return type

skll.types.CrossValidateTaskResults

Raises

- **ValueError** – If classification labels are not properly encoded as strings.
- **ValueError** – If `grid_search` is `True` but `grid_objective` is `None`.

evaluate(*examples*, *prediction_prefix=None*, *append=False*, *grid_objective=None*, *individual_predictions=False*, *output_metrics=[]*)

Evaluate the meta-estimator on a given `FeatureSet`.

Parameters

- **examples** (*skll.data.featureset.FeatureSet*) – The `FeatureSet` instance to evaluate the performance of the model on.
- **prediction_prefix** (*Optional[str], default=None*) – If saving the predictions, this is the prefix that will be used for the filename. It will be followed by `"_predictions.tsv"`
- **append** (*bool, default=False*) – Should we append the current predictions to the file if it exists?
- **grid_objective** (*Optional[str], default=None*) – The objective function used when doing the grid search.
- **individual_predictions** (*bool, default=False*) – Optionally, write out the predictions from each underlying learner.
- **output_metrics** (*List[str], default=[]*) – List of additional metric names to compute in addition to grid objective.

Returns

The confusion matrix, the overall accuracy, the per-label PRFs, the model parameters, the grid search objective function score, and the additional evaluation metrics, if any.

Return type

skll.types.EvaluateTaskResults

classmethod from_file(*learner_path*, *logger=None*)

Load a saved `VotingLearner` instance from a file.

Parameters

- **learner_path** (*skll.types.PathOrStr*) – The path to a saved `VotingLearner` instance file.
- **logger** (*Optional[logging.Logger], default=None*) – A logging object. If `None` is passed, get logger from `__name__`.

Returns

learner – The `VotingLearner` instance loaded from the file.

Return type

skll.learner.voting.VotingLearner

property learners: List[*Learner*]

Return the underlying list of learners.

learning_curve(*examples*, *metric*, *cv_folds=10*, *train_sizes=array([0.1, 0.325, 0.55, 0.775, 1.])*, *override_minimum=False*)

Generate learning curves for the meta-estimator.

Generate learning curves for the voting meta-estimator on the training examples via cross-validation. Adapted from the scikit-learn code for learning curve generation (cf.``sklearn.model_selection.learning_curve``).

Parameters

- **examples** (*skll.data.featureset.FeatureSet*) – The `FeatureSet` instance to generate the learning curve on.
- **metric** (*str*) – The name of the metric function to use when computing the train and test scores for the learning curve.
- **cv_folds** (Union[int, *skll.types.FoldMapping*], default=10) – The number of folds to use for cross-validation, or a mapping from example IDs to folds.
- **train_sizes** (*skll.types.LearningCurveSizes*, default= `numpy.linspace()` with start=0.1, stop=1.0, num=5) – Relative or absolute numbers of training examples that will be used to generate the learning curve. If the type is float, it is regarded as a fraction of the maximum size of the training set (that is determined by the selected validation method), i.e. it has to be within (0, 1]. Otherwise it is interpreted as absolute sizes of the training sets. Note that for classification the number of samples usually have to be big enough to contain at least one sample from each class.
- **override_minimum** (*bool, default=False*) – Learning curves can be unreliable for very small sizes esp. for > 2 labels. If this option is set to `True`, the learning curve would be generated even if the number of example is less 500 along with a warning. If `False`, the curve is not generated and an exception is raised instead.

Returns

- **train_scores** (*List[float]*) – The scores for the training set.
- **test_scores** (*List[float]*) – The scores on the test set.
- **fit_times** (*List[float]*) – The average times taken to fit each model.
- **num_examples** (*List[int]*) – The numbers of training examples used to generate the curve.

Raises
`ValueError` – If the number of examples is less than 500.

Return type
Tuple[*List*[float], *List*[float], *List*[float], *List*[int]]

property model

Return underlying scikit-learn meta-estimator model.

property model_type

Return meta-estimator model type (i.e., the class).

predict(*examples*, *prediction_prefix=None*, *append=False*, *class_labels=True*, *individual_predictions=False*)

Generate predictions with meta-estimator.

Compute the predictions from the meta-estimator and, optionally, the underlying estimators on given `FeatureSet`. The predictions are also written to disk if `prediction_prefix` is not `None`.

For regressors, the returned and written-out predictions are identical. However, for classifiers:

- if `class_labels` is `True`, class labels are returned as well as written out.
- if `class_labels` is `False` and the classifier is probabilistic (i.e., `self.probability` is `True`), class probabilities are returned as well as written out.
- if `class_labels` is `False` and the classifier is non-probabilistic (i.e., `self..probability` is `False`), class indices are returned and class labels are written out. This option is generally only meant for SKLL-internal use.

Parameters

- **examples** (*skll.data.featureset.FeatureSet*) – The `FeatureSet` instance to predict labels for.
- **prediction_prefix** (*Optional[str], default=None*) – If saving the predictions, this is the prefix that will be used for the filename. It will be followed by `"_predictions.tsv"`
- **append** (*bool, default=False*) – Should we append the current predictions to the file if it exists?
- **class_labels** (*bool, default=True*) – For classifier, should we convert class indices to their (str) labels for the returned array? Note that class labels are always written out to disk.
- **individual_predictions** (*bool, default=False*) – Return (and, optionally, write out) the predictions from each underlying learner.

Returns

The first element is the array of predictions returned by the meta-estimator and the second is an optional dictionary with the name of each underlying learner as the key and the array of its predictions as the value. The second element is `None` if `individual_predictions` is set to `False`.

Return type

Tuple[numpy.ndarray, Optional[Dict[str, numpy.ndarray]]]

`save`(*learner_path*)

Save the `VotingLearner` instance to a file.

Parameters

`learner_path` (*`skll.types.PathOrStr`*) – The path to save the `VotingLearner` instance to.

Return type

None

`train`(*examples*, *param_grid_list=None*, *grid_search_folds=5*, *grid_search=True*, *grid_objective=None*, *grid_jobs=None*, *shuffle=False*)

Train the voting meta-estimator.

First, we train each of the underlying estimators (represented by a sklI `Learner`), possibly with grid search. Then, we instantiate a `VotingClassifier` or `VotingRegressor` as appropriate with the scikit-learn `Pipeline` stored in the `pipeline` attribute of each trained `Learner` instance as the estimator. Finally, we call `fit()` on the `VotingClassifier` or `VotingRegressor` instance. We follow this process because it allows us to use grid search to find good hyperparameter values for our underlying learners before passing them to the meta-estimator AND because it allows us to use SKLL featuresets and do all of the same pre-processing when doing inference.

The trained meta-estimator is saved in the `_model` attribute. Nothing is returned.

Parameters

- **`examples`** (*`skll.data.featureset.FeatureSet`*) – The `FeatureSet` instance to use for training.
- **`param_grid_list`** (*`Optional[List[Dict[str, Any]]]`*, *`default=None`*) – The list of parameter grids to search through for grid search, one for each underlying learner. The order of the dictionaries should correspond to the order in which the underlying estimators were specified when the `VotingLearner` was instantiated. If `None`, the default parameter grids will be used for the underlying estimators.
- **`grid_search_folds`** (Union[int, *`skll.types.FoldMapping`*], default=5) – The number of folds to use when doing the grid search for

each of the underlying learners, or a mapping from example IDs to folds.

- **grid_search** (*bool, default=True*) – Should we use grid search when training each underlying learner?
- **grid_objective** (*Optional[str], default=None*) – The name of the objective function to use when doing the grid search for each underlying learner. Must be specified if `grid_search` is `True`.
- **grid_jobs** (*Optional[int], default=None*) – The number of jobs to run in parallel when doing the grid search for each underlying learner. If `None` or 0, the number of grid search folds will be used.
- **shuffle** (*bool, default=False*) – Shuffle examples (e.g., for grid search CV.)

Return type

None

1.7.6 `metrics` Module

Metrics that can be used to evaluate the performance of learners.

author

Nitin Madnani (nmadnani@ets.org)

author

Michael Heilman (mheilman@ets.org)

author

Dan Blanchard (dblanchard@ets.org)

organization

ETS

`sklll.metrics.`**`correlation`**(*y_true*, *y_pred*, *corr_type='pearson'*)

Calculate given correlation type between `y_true` and `y_pred`.

`y_pred` can be multi-dimensional. If `y_pred` is 1-dimensional, it may either contain probabilities, most-likely classification labels, or regressor predictions. In that case, we simply return the correlation between `y_true` and `y_pred`. If `y_pred` is multi-dimensional, it contains probabilties for multiple classes in which case, we infer the most likely labels and then compute the correlation between those and `y_true`.

Parameters

- **y_true** (*numpy.ndarray*) – The true/actual/gold labels for the data.
- **y_pred** (*numpy.ndarray*) – The predicted/observed labels for the data.

- **`corr_type`** (*`str`*, *`default="pearson"`*) – Which type of correlation to compute. Possible choices are "pearson", "spearman", and "kendall_tau".

Returns
correlation value if well-defined, else 0.0

Return type
float

`skll.metrics.`**`f1_score_least_frequent`**(*y_true*, *y_pred*)

Calculate F1 score of the least frequent label/class.

Parameters

- **`y_true`** (*`numpy.ndarray`*) – The true/actual/gold labels for the data.
- **`y_pred`** (*`numpy.ndarray`*) – The predicted/observed labels for the data.

Returns
F1 score of the least frequent label.

Return type
float

`skll.metrics.`**`kappa`**(*y_true*, *y_pred*, *weights=None*, *allow_off_by_one=False*)

Calculate the kappa inter-rater agreement.

The agreement is calculated between the gold standard and the predicted ratings. Potential values range from -1 (representing complete disagreement) to 1 (representing complete agreement). A kappa value of 0 is expected if all agreement is due to chance.

In the course of calculating kappa, all items in `y_true` and `y_pred` will first be converted to floats and then rounded to integers.

It is assumed that y_true and y_pred contain the complete range of possible ratings.

This function contains a combination of code from yorchopolis's kappa-stats and Ben Hamner's Metrics projects on Github.

Parameters

- **`y_true`** (*`numpy.ndarray`*) – The true/actual/gold labels for the data.
- **`y_pred`** (*`numpy.ndarray`*) – The predicted/observed labels for the data.
- **`weights`** (*`Optional[Union[str, numpy.ndarray]]`*, *`default=None`*) – Specifies the weight matrix for the calculation. Possible values are: `None` (unweighted-kappa), `"quadratic"` (quadratically weighted kappa), `"linear"` (linearly weighted kappa), and a two-dimensional numpy array (a custom matrix of weights). Each weight in this array corresponds to the w_{ij} values in the Wikipedia description of how to calculate weighted Cohen's kappa.

- **allow_off_by_one** (*bool, default=False*) – If true, ratings that are off by one are counted as equal, and all other differences are reduced by one. For example, 1 and 2 will be considered to be equal, whereas 1 and 3 will have a difference of 1 for when building the weights matrix.

Returns
The weighted or unweighted kappa score.

Return type
float

Raises

- **AssertionError** – If `y_true` != `y_pred`.
- **ValueError** – If labels cannot be converted to int.
- **ValueError** – If invalid weight scheme.

`sklearn.metrics.`**`register_custom_metric`**(*custom_metric_path*, *custom_metric_name*)

Import, load, and register the custom metric function from the given path.

Parameters

- **custom_metric_path** (*sklearn.types.PathOrStr*) – The path to a custom metric.
- **custom_metric_name** (*str*) – The name of the custom metric function to load. This function must take only two array-like arguments: the true labels and the predictions, in that order.

Raises

- **ValueError** – If the custom metric path does not end in '.py'.
- **NameError** – If the name of the custom metric file conflicts with an already existing attribute in `skll.metrics` or if the custom metric name conflicts with a scikit-learn or SKLL metric.

`skll.metrics.`**`use_score_func`**(*func_name*, *y_true*, *y_pred*)

Call the given scoring function.

This takes care of handling keyword arguments that were pre-specified when creating the scorer. This applies any sign-flipping that was specified by `make_scorer()` when the scorer was created.

Parameters

- **func_name** (*str*) – The name of the objective function to use.
- **y_true** (*numpy.ndarray*) – The true/actual/gold labels for the data.
- **y_pred** (*numpy.ndarray*) – The predicted/observed labels for the data.

Returns
The scored result from the given scorer.

Return type
float

1.7.7 `utils` Package

Various useful constants defining groups of evaluation metrics.

`sklearn.utils.constants.`**`CLASSIFICATION_ONLY_METRICS = {'accuracy', 'average_precision', 'balanced_accuracy', 'f05', 'f05_score_macro', 'f05_score_micro', 'f05_score_weighted', 'f1', 'f1_score_least_frequent', 'f1_score_macro', 'f1_score_micro', 'f1_score_weighted', 'jaccard', 'jaccard_macro', 'jaccard_micro', 'jaccard_weighted', 'neg_log_loss', 'precision', 'precision_macro', 'precision_micro', 'precision_weighted', 'recall', 'recall_macro', 'recall_micro', 'recall_weighted', 'roc_auc'}`**

Set of evaluation metrics only used for classification tasks

`skll.utils.constants.`**`CORRELATION_METRICS = {'kendall_tau', 'pearson', 'spearman'}`**

Set of evaluation metrics based on correlation

`skll.utils.constants.`**`PROBABILISTIC_METRICS = frozenset({'average_precision', 'neg_log_loss', 'roc_auc'})`**

Set of evaluation metrics that can use prediction probabilities

`skll.utils.constants.`**`REGRESSION_ONLY_METRICS = {'explained_variance', 'max_error', 'neg_mean_absolute_error', 'neg_mean_squared_error', 'neg_root_mean_squared_error', 'r2'}`**

Set of evaluation metrics only used for regression tasks

`skll.utils.constants.`**`UNWEIGHTED_KAPPA_METRICS = {'unweighted_kappa', 'uwk_off_by_one'}`**

Set of unweighted kappa agreement metrics

`skll.utils.constants.`**`WEIGHTED_KAPPA_METRICS = {'linear_weighted_kappa', 'lwk_off_by_one', 'quadratic_weighted_kappa', 'qwk_off_by_one'}`**

Set of weighed kappa agreement metrics

A useful logging function for SKLL developers

`skll.utils.logging.`**`get_skll_logger`**(*name*, *filepath=None*, *log_level=20*)

Create and return logger instances appropriate for use in SKLL code.

These logger instances can log to both STDERR as well as a file. This function will try to reuse any previously created logger based on the given name and filepath.

Parameters

- **name** (*str*) – The name to be used for the logger.
- **filepath** (*Optional[str], default=None*) – The file to be used for the logger via a FileHandler. Default: None in which case no file is attached to the logger.
- **log_level** (*int, default=logging.INFO*) – The level for logging messages

Returns
logger – A `Logger` instance.

Return type
logging.Logger

1.7.8 types Module

The `skll.types` module contains custom type aliases that are used throughout the SKLL code in type hints and docstrings.

`skll.types.`**`ClassMap`**
alias of `Dict`[`str`, `List`[`str`]]

Class map that maps new labels (string) to list of old labels (list of string).

`skll.types.`**`ConfusionMatrix`**
alias of `List`[`List`[`int`]]

Confusion matrix represented by a list of list of integers.

`skll.types.`**`FeatureDict`**
alias of `Dict`[`str`, `Any`]

Feature dictionary that maps a string to other dictionaries or other objects.

`skll.types.`**`FeatureDictList`**
alias of `List`[`Dict`[`str`, `Any`]]

List of feature dictionaries.

`skll.types.`**`FeaturesetIterator`**
alias of `Iterator`[`Tuple`[`skll.data.FeatureSet`, `skll.data.FeatureSet`]]

An iterator over two FeatureSets, usually test and train.

`skll.types.`**`FoldMapping`**
alias of `Dict`[`Union`[`float`, `str`], `str`]

Mapping from example ID to fold ID; the example ID may be a float or a string but the fold ID is always a string.

`skll.types.`**`IdType`**

  alias of `Union[float, str]`

A float or a string; this is useful or SKLL IDs that can be both.

`skll.types.`**`IndexIterator`**

  alias of `Generator[Tuple[ndarray, ndarray], None, None]`

Generator over two `numpy` arrays containing indices - usually for train and test data.

`skll.types.`**`LabelType`**

  alias of `Union[float, int, str]`

A float, integer, or a string; this is useful for SKLL labels that can be any of them.

`skll.types.`**`LearningCurveSizes`**

  alias of `Union[List[Union[float, int]], ndarray]`

Learning curve sizes can either be a `numpy` array (or a list) containing floats or integers.

`skll.types.`**`FeatGenerator`**

  alias of `Generator[Tuple[Union[float, str], Optional[Union[float, int, str]], Dict[str, Any]], None, None]`

Generator that yields a 3-tuple containing:

1. An example ID (float or string).
2. A label (integer, float, or string).
3. A feature dictionary.

`skll.types.`**`PathOrStr`**

  alias of `Union[Path, str]`

A string path or Path object.

`skll.types.`**`SparseFeatureMatrix`**

  alias of `csr_matrix`

A `scipy` sparse matrix to hold SKLL features in FeatureSets.

`skll.types.`**`ComputeEvalMetricsResults`**

  alias of `Tuple[Optional[List[List[int]]], Optional[float], Dict[Union[float, int, str], Any], Optional[float], Dict[str, Optional[float]]]`

Learner evaluate task results 5-tuple containing:

1. The confusion matrix for a classifier, `None` for a regressor.
2. Accuracy for a classifier, `None` for a regressor.

3. The dictionary of results.
4. Score for the grid objective, `None` if no grid search was performed.
5. The dictionary of scores for any additional metrics.

`skll.types.`**`EvaluateTaskResults`**

alias of `Tuple[Optional[List[List[int]]], Optional[float], Dict[Union[float, int, str], Any], Dict[str, Any], Optional[float], Dict[str, Optional[float]]]`

Learner evaluate task results 6-tuple containing:

1. The confusion matrix for a classifier, `None` for a regressor.
2. Accuracy for a classifier, `None` for a regressor.
3. The dictionary of results.
4. The dictionary containing the model parameters.
5. Score for the grid objective, None if no grid search
6. The dictionary of score for any additional metrics.

`skll.types.`**`CrossValidateTaskResults`**

alias of `Tuple[List[Tuple[Optional[List[List[int]]], Optional[float], Dict[Union[float, int, str], Any], Dict[str, Any], Optional[float], Dict[str, Optional[float]]]], List[float], List[Dict[str, Any]], Optional[Dict[Union[float, str], str]], Optional[List[`*`skll.learner.Learner`*`]]]`

Learner cross-validate task results 5-tuple containing:

1. The confusion matrix, overall accuracy, per-label precision/recall/F1, model parameters, objective function score, and evaluation metrics (if any) for each fold.
2. The grid search scores for each fold.
3. The list of dictionaries of grid search CV results, one per fold, with keys such as "params", "mean_test_score", etc, that are mapped to lists of values associated with each combination of hyper-parameters.
4. The dictionary containing the test-fold number for each, `None` if folds were not saved.
5. The list of learners, one for each fold, `None` if the models were not saved.

`skll.types.`**`VotingCrossValidateTaskResults`**

alias of `Tuple[List[Tuple[Optional[List[List[int]]], Optional[float], Dict[Union[float, int, str], Any], Dict[str, Any], Optional[float], Dict[str, Optional[float]]]], Optional[Dict[Union[float, str], str]], Optional[List[`*`skll.learner.voting.VotingLearner`*`]]]`

Voting Learner cross-validate task results 3-tuple containing:

1. The confusion matrix, overall accuracy, per-label precision/recall/F1, model parameters, objective function score, and evaluation metrics (if any) for each fold.
2. The dictionary containing the test-fold number for each, `None` if folds were not saved.
3. The list of voting learners, one for each fold, `None` if the models were not saved.

1.8 Contributing

Thank you for your interest in contributing to SKLL! We welcome any and all contributions.

1.8.1 Guidelines

The SKLL contribution guidelines can be found in our Github repository here. We strongly encourage all SKLL contributions to follow these guidelines.

1.8.2 SKLL Code Overview

This section will help you get oriented with the SKLL codebase by describing how it is organized, the various SKLL entry points into the code, and what the general code flow looks like for each entry point.

Organization

The main Python code for the SKLL package lives inside the `skll` sub-directory of the repository. It contains the following files and sub-directories:

- config/ : Code to parse SKLL experiment configuration files.
- experiments/ : Code that is related to creating and running SKLL experiments. It also contains code that collects the various evaluation metrics and predictions for each SKLL experiment and writes them out to disk.
- learner/ : Code for the Learner and VotingLearner classes. The former is instantiated for all learner names specified in the experiment configuration file *except* `VotingClassifier` and `VotingRegressor` for which the latter is instantiated instead.
- metrics.py : Code for any custom metrics that are not in `sklearn.metrics`, e.g., `kappa`, `kendall_tau`, `spearman`, etc. This module also contains the code that powers *user-defined custom metrics*.
- data/
  - __init__.py : Code used to initialize the `skll.data` Python package.

  - featureset.py : Code for the `FeatureSet` class metadata for a given set of instances.
  - readers.py : Code for classes that can read various file formats and create `FeatureSet` objects from them.
  - writers.py : Code for classes that can write `FeatureSet` objects to files on disk in various formats.
  - dict_vectorizer.py : Code for a `DictVectorizer` class that subclasses `sklearn.feature_extraction.DictVectorizer` to add an `__eq__()` method that we need for vectorizer equality.
- utils/ : Code for different utility scripts, functions, and classes used throughout SKLL. The most important ones are the command line scripts in the `utils.commandline` submodule.
  - compute_eval_from_predictions.py : See documentation.
  - filter_features.py : See documentation.
  - generate_predictions.py : See documentation.
  - join_features.py : See documentation.
  - plot_learning_curves.py : See documentation.
  - print_model_weights.py : See documentation.
  - run_experiment.py : See documentation.
  - skll_convert.py : See documentation.
  - summarize_results.py : See documentation.
- version.py : Code to define the SKLL version. Only changed for new releases.
- tests/ - `test_*.py` : These files contain the code for the unit tests and regression tests.

Entry Points & Workflow

There are three main entry points into the SKLL codebase:

1. **Experiment configuration files**. The primary way to interact with SKLL is by writing configuration files and then passing it to the run_experiment script. When you run the command `run_experiment <config_file>`, the following happens (at a high level):
   - the configuration file is handed off to the run_configuration() function in `experiments.py`.
   - a SKLLConfigParser object is instantiated from `config.py` that parses all of the relevant fields out of the given configuration file.
   - the configuration fields are then passed to the _classify_featureset() function in `experiments.py` which instantiates the learners (using code from `learner.py`), the

featuresets (using code from `reader.py` & `featureset.py`), and runs the experiments, collects the results, and writes them out to disk.

2. **SKLL API**. Another way to interact with SKLL is via the SKLL API directly in your Python code rather than using configuration files. For example, you could use the Learner.from_file() or VotingLearner.from_file() methods to load saved models of those types from disk and make predictions on new data. The documentation for the SKLL API can be found here.
3. **Utility scripts**. The scripts listed in the section above under `utils` are also entry points into the SKLL code. These scripts are convenient wrappers that use the SKLL API for commonly used tasks, e.g., generating predictions on new data from an already trained model.

1.9 Internal Documentation

1.9.1 Release Process

This document is only meant for the project administrators, not users and developers.

1. Create a release branch `release/XX` on GitHub.
2. In the release branch:
   a. Update the version numbers in `version.py`.
   b. Make sure that *requirements.txt* only has the actual dependencies that are needed to run SKLL. Any dependencies needed only for development/testing (e.g., *sphinx*, *nose2* etc.) should be moved to *requirements.dev*. This means that *requirements.txt must* be a strict subset of *requirements.dev*.
   c. Make sure the versions in *doc/requirements.txt* are up to date with *requirements.txt* and only contains the dependencies needed to build the documentation.
   d. Make sure *.readthedocs.yml* is still accurate.
   e. Update the conda recipe.
   f. Update the documentation with any new features or details about changes.
   g. Run `make linkcheck` on the documentation and fix any redirected/broken links.
   h. Update the README and this release documentation, if necessary.
3. Build and upload the conda packages by following instructions in `conda-recipe/README.md`.
4. Build the PyPI source distribution using `python setup.py sdist build`.
5. Upload the source distribution to TestPyPI using `twine upload --repository testpypi dist/*`. You will need to have the `twine` package installed and set up your `$HOME/.pypirc` correctly. See details here.

6. Test the conda package by creating a new environment on different platforms with this package installed and then running SKLL examples or tests from a SKLL working copy. If the package works, then move on to the next step. If it doesn't, figure out why and rebuild and re-upload the package.

7. Test the TestPyPI package by installing it as follows:

```
pip install --index-url https://test.pypi.org/simple/ --extra-index-
↪url https://pypi.org/simple skll
```

8. Then run some SKLL examples or tests from a SKLL working copy. If the TestPyPI package works, then move on to the next step. If it doesn't, figure out why and rebuild and re-upload the package.

9. Create pull requests on the skll-conda-tester and skll-pip-tester repositories to test the conda and TestPyPI packages on Linux and Windows.

10. Draft a release on GitHub while the Linux and Windows package tester builds are running.

11. Once both builds have passed, make a pull request with the release branch to be merged into `main` and request code review.

12. Once the build for the PR passes and the reviewers approve, merge the release branch into `main`.

13. Upload source and wheel packages to PyPI using `python setup.py sdist upload` and `python setup.py bdist_wheel upload`

14. Make sure that the ReadTheDocs build for `main` passes.

15. Tag the latest commit in `main` with the appropriate release tag and publish the release on GitHub.

16. Send an email around at ETS announcing the release and the changes.

17. Post release announcement on Twitter/LinkedIn.

CHAPTER

TWO

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

INDEX