SciKit-Learn Laboratory Documentation

Release 1.5.3

Daniel Blanchard   **Michael Heilman**
Nitin Madnani

Jun 20, 2019

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 1

Documentation

1.1 Installation

SKLL can be installed via `pip` for any Python version:

```
pip install skll
```

or via `conda` (only for Python 3.6):

```
conda install -c defaults -c conda-forge -c desilinguist python=3.6␣
↪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 results

1.3.2 Titanic Example

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

Get your data into the correct format

The first step to getting the Titanic data is logging into Kaggle and downloading train.csv and test.csv. Once you have those files, you'll also want to grab the examples folder on our GitHub page and put `train.csv` and `test.csv` in `examples`.

The provided script, `make_titanic_example_data.py`, will split the training 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
objectives = ['accuracy']

[Output]
# again, these can be absolute paths
metrics = ['roc_auc']
log = 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. *log* 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
```

That should produce output like:

```
2017-12-07 11:40:17,381 - Titanic_Evaluate_Tuned_family.csv+misc.
↪csv+socioeconomic.csv+vitals.csv_RandomForestClassifier - INFO -␣
↪Task: evaluate
2017-12-07 11:40:17,381 - 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
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
2017-12-07 11:40:17,515 - Titanic_Evaluate_Tuned_family.csv+misc.
↪csv+socioeconomic.csv+vitals.csv_RandomForestClassifier - INFO -␣
↪Featurizing and training new RandomForestClassifier model
2017-12-07 11:40:17,515 - 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.
```

(continues on next page)

(continued from previous page)

```
2017-12-07 11:40:21,650 - Titanic_Evaluate_Tuned_family.csv+misc.
↪csv+socioeconomic.csv+vitals.csv_RandomForestClassifier - INFO -␣
↪Best accuracy grid search score: 0.809
2017-12-07 11:40:21,651 - Titanic_Evaluate_Tuned_family.csv+misc.
↪csv+socioeconomic.csv+vitals.csv_RandomForestClassifier - INFO -␣
↪Hyperparameters: bootstrap: True, class_weight: None, criterion:␣
↪gini, max_depth: 10, max_features: auto, 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, n_
↪estimators: 500, n_jobs: 1, oob_score: False, random_state:␣
↪123456789, verbose: 0, warm_start: False
2017-12-07 11:40:21,651 - Titanic_Evaluate_Tuned_family.csv+misc.
↪csv+socioeconomic.csv+vitals.csv_RandomForestClassifier - INFO -␣
↪Evaluating predictions
```

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 config file is:

```
Experiment Name: Titanic_Evaluate_Tuned
SKLL Version: 1.5
Training Set: train
Training Set Size: 712
Test Set: dev
Test Set Size: 179
Shuffle: False
```

(continues on next page)

(continued from previous page)

```
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: 0.19.1
Start Timestamp: 07 Dec 2017 11:42:04.911657
End Timestamp: 07 Dec 2017 11:42:09.118036
Total Time: 0:00:04.206379


Fold:
Model Parameters: {"bootstrap": true, "class_weight": null, "criterion
↪": "gini", "max_depth": 10, "max_features": "auto", "max_leaf_nodes
↪": 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": 1, "oob_score": false,
↪"random_state": 123456789, "verbose": 0, "warm_start": false}
Grid Objective Score (Train) = 0.8089887640449438
+---+-------+------+-----------+--------+-----------+
|   |     0 |    1 | Precision | Recall | F-measure |
+---+-------+------+-----------+--------+-----------+
| 0 | [101] |   14 |     0.871 |  0.878 |     0.874 |
+---+-------+------+-----------+--------+-----------+
| 1 |    15 | [49] |     0.778 |  0.766 |     0.772 |
+---+-------+------+-----------+--------+-----------+
(row = reference; column = predicted)
Accuracy = 0.8379888268156425
Objective Function Score (Test) = 0.8379888268156425

Additional Evaluation Metrics (Test):
 roc_auc = 0.8219429347826087
```

1.3.3 Running your own experiments

Once you've gone through the Titanic example, you will hopefully be interested in trying out SKLL with your own data. To do so, you'll still need to get your data in an appropriate format first.

Get your data into the correct format

Supported formats

SKLL can work with several common data formats, each 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
```

Creating sparse files

skll_convert can also create sparse data files in *.jsonlines*, *.libsvm*, *.megam*, 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.

Training and testing directories

At 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.

Single-file training and testing sets

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.

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

Creating a configuration file

Now that you've seen a *basic configuration file*, you should look at the extensive option available in our *config file reference*.

Running your 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*.

1.4 Running Experiments

The simplest way to use SKLL is to create configuration files that describe experiments you would like to run on pre-generated features. This document describes the supported feature file formats, how to create configuration files (and layout your directories), and how to use *run_experiment* to get things going.

1.4.1 Quick Example

If you don't want to read the whole document, and just want an example of how things work, do the following from the command prompt:

```
$ cd examples
$ python make_iris_example_data.py          # download a simple dataset
$ cd iris
$ run_experiment --local evaluate.cfg        # run an experiment
```

1.4.2 Feature file formats

The following feature file formats are supported:

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 with the following restrictions:

- If the data is labelled, there must be a column with the name specified by *label_col* in the *Input* section of the configuartion 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).

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: `|` `#` `=`

megam

An expanded form of the input format for the MegaM classification package with the `-fvals` switch.

The basic format is:

```
# Instance1
CLASS1    F0 2.5 F1 3 FEATURE_2 -152000
# Instance2
CLASS2    F1 7.524
```

where the **optional** comments before each instance specify the ID for the following line, class names are separated from feature-value pairs with a tab, and feature-value pairs are separated by spaces. Any omitted features for a given instance are assumed to be zero, so this format is handy when dealing with sparse data. We also include several utility scripts for converting to/from this MegaM format and for adding/removing features from the files.

1.4.3 Creating configuration files

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 possible settings for 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`.

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

  Note: 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.

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

Example configuration files are available here.

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 has only one required field, *learners*, but also must contain either *train_file* or *train_directory*.

learners

List of scikit-learn models to try using. A separate job will be run for each combination of classifier and feature-set. 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 a `base_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 require setting an additional `algorithm` fixed parameter with the value `'SAMME'`.
- **DummyClassifier**: Simple rule-based Classification
- **DecisionTreeClassifier**: Decision Tree Classification
- **GradientBoostingClassifier**: Gradient Boosting Classification
- **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

Regressors:

- **AdaBoostRegressor**: AdaBoost Regression. Note that the default base estimator is a `DecisionTreeRegressor`. A different base estimator can be used by specifying a `base_estimator` fixed parameter in the *fixed_parameters* list. The following additional base estimators are supported: `SGDRegressor`, and `SVR`.
- **BayesianRidge**: Bayesian Ridge Regression
- **DecisionTreeRegressor**: Decision Tree Regressor
- **DummyRegressor**: Simple Rule-based Regression
- **ElasticNet**: ElasticNet Regression
- **GradientBoostingRegressor**: Gradient Boosting Regressor
- **HuberRegressor**: Huber Regression
- **KNeighborsRegressor**: K-Nearest Neighbors Regression
- **Lars**: Least Angle Regression

- **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 `base_estimator` fixed parameter in the *fixed_parameters* list.
- **Ridge**: Ridge Regression
- **SGDRegressor**: Stochastic Gradient Descent Regression
- **SVR**: Support Vector Regression using LibSVM
- **TheilSenRegressor**: Theil-Sen Regression

For all regressors 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.

train_file *(Optional)*

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 *(Optional)*

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.

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*.

featuresets *(Optional)*

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.

suffix *(Optional)*

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

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`.

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).

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`).

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`.

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.

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`.

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]

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

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, if you specify the following learners: `["SVC", "LogisticRegression"]`, specifying `[10, 100]` as the value of `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 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 have 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]`.

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`.

sampler *(Optional)*

It performs a 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.

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}
```

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.

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.

featureset_names *(Optional)*

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

fixed_parameters *(Optional)*

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

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

AdaBoostClassifier and AdaBoostRegressor

```
{'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}
```

Lasso:

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

LinearSVC and LinearSVR

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

LogisticRegression

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

MLPClassifier and MLPRegressor:

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

RandomForestClassifier and RandomForestRegressor

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

RANSACRegressor

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

Ridge and RidgeClassifier

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

SVC and SVR

```
{'cache_size': 1000}
```

SGDClassifier

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

SGDRegressor

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

TheilSenRegressor

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

Note: This option allows us to deal with imbalanced data sets by using the parameter `class_weight` for the classifiers: `DecisionTreeClassifier`, `LogisticRegression`, `LinearSVC`, `RandomForestClassifier`, `RidgeClassifier`, `SGDClassifier`, and `SVC`.

Two possible options are available. The first one is `balanced`, which automatically adjust 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.

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`.

Tuning

grid_search *(Optional)*

Whether or not to perform grid search to find optimal parameters for classifier. Defaults to `False`. Note that for the *learning_curve* task, grid search is not allowed and setting it to `True` will generate a warning and be ignored.

grid_search_folds *(Optional)*

The number of folds to use for grid search. Defaults to 3.

grid_search_jobs *(Optional)*

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

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.

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.

objectives *(Optional)*

The objective functions to use for tuning. This is a list of one or more objective functions. Valid options are:

Classification:

- **accuracy**: Overall accuracy
- **precision**: Precision
- **recall**: Recall
- **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_micro**: Micro-averaged F_1 score
- **f1_score_macro**: Macro-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.
- **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 as the objective, *probability* must be set to `True`.
- **average_precision**: Area under PR curve (for binary classification)
- **roc_auc**: Area under ROC curve (for binary classification)

Regression or classification with integer labels:

- **unweighted_kappa**: Unweighted Cohen's kappa (any floating point values are rounded to ints)
- **linear_weighted_kappa**: Linear weighted kappa (any floating point values are rounded to ints)
- **quadratic_weighted_kappa**: Quadratic weighted 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.
- **lwk_off_by_one**: Same as `linear_weighted_kappa`, but all ranking differences are discounted by one.
- **qwk_off_by_one**: Same as `quadratic_weighted_kappa`, but all ranking differences are discounted by one.

Regression or classification with binary labels:

- **kendall_tau**: Kendall's tau
- **pearson**: Pearson correlation
- **spearman**: Spearman rank-correlation

Regression:

- **r2**: R2
- **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.

Defaults to `['f1_score_micro']`.

Note:

1. Using `objective=x` instead of `objectives=['x']` is also acceptable, for backward-compatibility.
2. Also see the *metrics* option below.

---

param_grids *(Optional)*

List of parameter grids to search for each learner. Each parameter grid should be a list of dictionaries mapping from strings to lists of parameter values. When you specify an empty list 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]}]
```

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': ["auto", None]}]
```

ElasticNet

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

GradientBoostingClassifier and GradientBoostingRegressor

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

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: Note that learners not listed here do not have any default parameter grids in SKLL either because either 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.

pos_label_str *(Optional)*

The string label for the positive class in the binary classification setting. If unspecified, an arbitrary class is picked.

Output

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.

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 import Learner
from skll.data import Reader

# 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/boston/train/example_boston_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": 0.09178, "f1": 0.0, "f2": 4.05, "f3": 0.0, "f4": 0.51, "f5
↪": 6.416, "f6": 84.1, "f7": 2.6463, "f8": 5.0, "f9": 296.0, "f10":␣
↪16.6, "f11": 395.5, "f12": 9.04}
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)))))
```

(continues on next page)

(continued from previous page)

```
```

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.

results *(Optional)*

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

metrics *(Optional)*

For the `evaluate` and `cross_validate` tasks, this is a list of additional metrics that will be computed *in addition to* the tuning objectives and added to the results files. For the `learning_curve` task, this will be the list of metrics for which the learning curves will be plotted. Can take all of the same functions as those available for the *tuning objectives*.

Note:

1. For learning curves, `metrics` can be specified instead of `objectives` since both serve the same purpose. If both are specified, `objectives` will be ignored.
2. For the `evaluate` and `cross_validate` tasks, any functions that are specified in both `metrics` and `objectives` are assumed to be the latter.
3. If you just want to use `neg_log_loss` as an additional metric, you do not need to set *probability* to `True`. That's only neeeded for `neg_log_loss` to be used as a tuning objective.

log *(Optional)*

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

models *(Optional)*

Directory to store trained models in. Can be omitted to not store models.

predictions *(Optional)*

Directory to store prediction files in. Can be omitted to not store predictions.

Note: You can use the same directory for *results*, *log*, *models*, and *predictions*.

save_cv_folds *(Optional)*

Whether to save the folds that were used for a cross-validation experiment to a CSV file named `EXPERIMENT_skll_fold_ids.csv` in the *results (Optional)* directory, where `EXPERIMENT` refers to the *experiment_name*. Defaults to `False`.

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:

-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.

[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.

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]

-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`).

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

Output files

For most of the tasks, the result, log, model, and prediction files generated by `run_experiment` will all share the automatically generated prefix `EXPERIMENT_FEATURESET_LEARNER_OBJECTIVE`, where the following definitions hold:

EXPERIMENT The name specified as *experiment_name* in the configuration file.

FEATURESET The feature set we're training on joined with "+".

LEARNER The learner the current results/model/etc. was generated using.

OBJECTIVE The objective function the current results/model/etc. was generated using.

However, if `objectives` contains only one objective function, the result, log, model, and prediction files will share the prefix `EXPERIMENT_FEATURESET_LEARNER`. For backward-compatibility, the same applies when a single objective is specified using `objective=x`.

In addition to the above log files that are specific to each "job" (a specific combination of featuresets, learners, and objectives specified in the configuration file), SKLL also produces a single, top level "experiment" log file with only `EXPERIMENT` as the prefix. While the job-level log files contain messages that pertain to the specific characteristics of the job, the experiment-level log file will contain logging messages that pertain to the overall experiment and configuration file. The messages in the log files are in the following format:

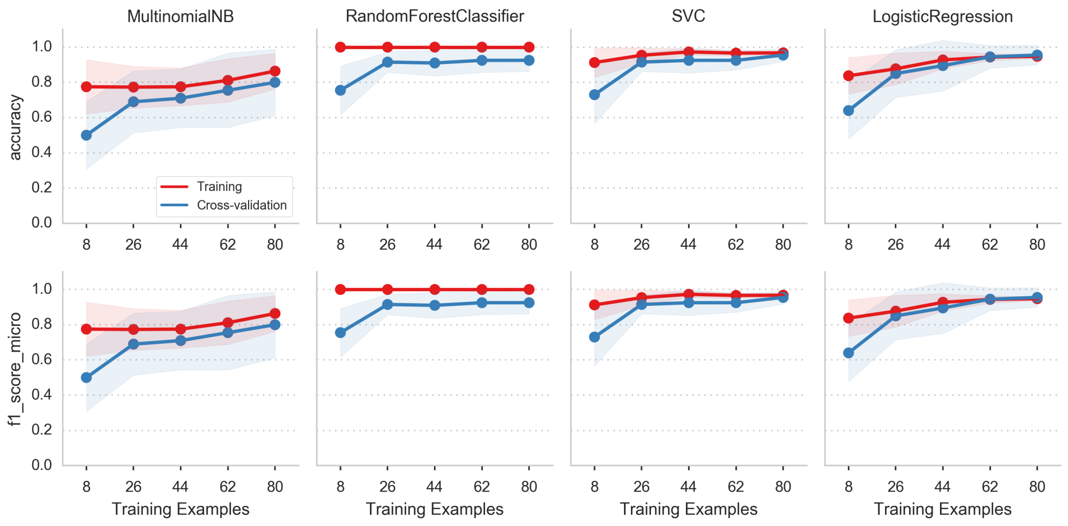
```
TIMESTAMP - LEVEL - MSG
```

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.

For every experiment you run, there will also be a result summary file generated that is a tab-delimited file summarizing the results for each learner-featureset combination you have in your configuration file. 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.

If seaborn is available when running a *learning_curve* experiment, actual learning curves are also generated as PNG files - one for each feature set specified in the configuration file. Each PNG file is named `EXPERIMENT_FEATURESET.png` and contains a faceted learning curve plot for the featureset with objective functions on rows and learners on columns. Here's an example of such a plot.

If you didn't have seaborn available when running the learning curve experiment, you can always generate the plots later from the learning curve summary file using the *plot_learning_curves* utility script.



1.5 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.

1.5.1 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.

1.5.2 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.

Positional Arguments

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

`outfile`
: 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.

`-i, --inverse`
: Instead of keeping features and/or examples in lists, remove them.

`-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`)

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

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

1.5.3 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.

Positional Arguments

`model_file`
Model file to load and use for generating predictions.

`input_file(s)`
One or more csv file(s), jsonlines file(s), or megam file(s) (with or without the label column), with the appropriate suffix.

Optional Arguments

`-a, --all_probabilities`
Flag indicating whether to output the probabilities of all labels instead of just the probability of the positive label.

`-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`)

`-p` `<positive_label>`, **`--positive_label`** `<positive_label>`
If the model is only being used to predict the probability of a particular label, this specifies the index of the label we're predicting. 1 = second label, which is default for binary classification. Keep in mind that labels are sorted lexicographically. (default: 1)

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

`-t` `<threshold>`, **`--threshold`** `<threshold>`
If the model we're using is generating probabilities of the positive label, return 1 if it meets/exceeds the given threshold and 0 otherwise.

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

1.5.4 join_features

Combine multiple feature files into one larger file.

Positional Arguments

`infile ...`
Input feature files (ends in `.arff`, `.csv`, `.jsonlines`, `.megam`, `.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.

1.5.5 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.

1.5.6 print_model_weights

Prints out the weights of a given trained model.

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`)

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

1.5.7 skll_convert

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

Positional Arguments

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

`outfile`
: Output feature file (ends in `.arff`, `.csv`, `.jsonlines`, `.libsvm`, `.megam`, `.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`)

`--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.

1.5.8 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.

1.6 API Documentation

1.6.1 Quickstart

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

Load a `FeatureSet` from a file:

```
from skll import Reader

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

Or, work with an existing `pandas DataFrame`:

```
from skll import FeatureSet

train_examples = FeatureSet.from_data_frame(my_data_frame, 'A Name for␣
↪My Data', labels_column='name of the column containing the data␣
↪labels')
```

(continues on next page)

(continued from previous page)

```
```

Train a linear svm (assuming we have `train_examples`):

```
from skll 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)
```

1.6.2 `skll` Package

The most useful parts of our API are available at the package level in addition to the module level. They are documented in both places for convenience.

From `data` Package

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

Bases: `object`

Encapsulation of all of the features, values, and metadata about a given set of data. This replaces *ExamplesTuple* from older versions of SKLL.

Parameters

- **name** (*str*) – The name of this feature set.

- **`ids`** (*`np.array`*) – Example IDs for this set.
- **`labels`** (*`np.array, optional`*) – labels for this set. Defaults to `None`.
- **`feature`** (*`list of dict or array-like, optional`*) – The features for each instance represented as either a list of dictionaries or an array-like (if *vectorizer* is also specified). Defaults to `None`.
- **`vectorizer`** (*`DictVectorizer or FeatureHasher, optional`*) – Vectorizer which will be used to generate the feature matrix. Defaults to `None`.

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*)

Removes or keeps features and/or examples from the *Featureset* depending on the parameters. Filtering is done in-place.

Parameters

- **`ids`** (*`list of str/float, optional`*) – Examples to keep in the FeatureSet. If *None*, no ID filtering takes place. Defaults to `None`.
- **`labels`** (*`list of str/float, optional`*) – Labels that we want to retain examples for. If *None*, no label filtering takes place. Defaults to `None`.
- **`features`** (*`list of str, optional`*) – 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. Defaults to `None`.
- **`inverse`** (*`bool, optional`*) – Instead of keeping features and/or examples in lists, remove them. Defaults to `False`.

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

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

A version of ***__iter__*** that retains only the specified features and/or examples from the output.

Parameters

- **`ids`** (*`list of str/float, optional`*) – Examples to keep in the `FeatureSet`. If `None`, no ID filtering takes place. Defaults to `None`.
- **`labels`** (*`list of str/float, optional`*) – Labels that we want to retain examples for. If `None`, no label filtering takes place. Defaults to `None`.
- **`features`** (*`list of str, optional`*) – 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. Defaults to `None`.
- **`inverse`** (*`bool, optional`*) – Instead of keeping features and/or examples in lists, remove them. Defaults to `False`.

Yields

- **id_** (*str*) – The ID of the example.
- **label_** (*str*) – The label of the example.
- **feat_dict** (*dict*) – The feature dictionary, with feature name as the key and example value as the value.

Raises `ValueError` – If the vectorizer is not a *DictVectorizer*.

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

Helper function to create a `FeatureSet` instance from a *pandas.DataFrame*. 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`** (*`pd.DataFrame`*) – The pandas.DataFrame object to use as a `FeatureSet`.
- **`name`** (*`str`*) – The name of the output `FeatureSet` instance.
- **`labels_column`** (*`str, optional`*) – The name of the column containing the labels (data to predict). Defaults to `None`.

- **vectorizer** (*DictVectorizer or FeatureHasher, optional*) – Vectorizer which will be used to generate the feature matrix. Defaults to `None`.

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

Return type *skll.FeatureSet*

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_by_ids (*fs*, *ids_for_split1*, *ids_for_split2=None*)

Split the `FeatureSet` into two new `FeatureSet` instances based on the given IDs for the two splits.

Parameters

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

Returns

- **fs1** (*skll.FeatureSet*) – The first `FeatureSet`.
- **fs2** (*skll.FeatureSet*) – The second `FeatureSet`.

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

Bases: `object`

A helper class to make picklable iterators out of example dictionary generators.

Parameters

- **path_or_list** (*str or list of dict*) – Path or a list of example dictionaries.

- **quiet** (*bool, optional*) – Do not print "Loading…" status message to stderr. Defaults to `True`.
- **ids_to_floats** (*bool, optional*) – Convert IDs to float to save memory. Will raise error if we encounter an a non-numeric ID. Defaults to `False`.
- **label_col** (*str, optional*) – 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. Defaults to `'y'`.
- **id_col** (*str, optional*) – 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. Defaults to `'id'`.
- **class_map** (*dict, optional*) – 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. Defaults to `None`.
- **sparse** (*bool, optional*) – Whether or not to store the features in a numpy CSR matrix when using a DictVectorizer to vectorize the features. Defaults to `True`.
- **feature_hasher** (*bool, optional*) – Whether or not a FeatureHasher should be used to vectorize the features. Defaults to `False`.
- **num_features** (*int, optional*) – 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. Defaults to `None`.
- **logger** (*logging.Logger, optional*) – A logger instance to use to log messages instead of creating a new one by default. Defaults to `None`.

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

Instantiate the appropriate Reader sub-class based on the file extension of the given path. Or use a dictionary reader if the input is a list of dictionaries.

Parameters

- **path_or_list** (*str or list of dicts*) – A path or list of example dictionaries.
- **kwargs** (*dict, optional*) – 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.Reader*

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

read()

Loads examples in the *.arff*, *.csv*, *.jsonlines*, *.libsvm*, *.megam*, *.ndj*, or *.tsv* formats.

Returns **feature_set** – `FeatureSet` instance representing the input file.

Return type *skll.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 skll.**Writer**(*path*, *feature_set*, ***kwargs*)

Bases: `object`

Helper class for writing out FeatureSets to files on disk.

Parameters

- **path** (*str*) – A path to the feature file we would like to create. The suffix to this filename must be `.arff`, `.csv`, `.jsonlines`, `.libsvm`, `.megam`, `.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.FeatureSet`) – The `FeatureSet` instance to dump to the file.
- **quiet** (*bool*) – Do not print "Writing..." status message to stderr. Defaults to `True`.
- **requires_binary** (*bool*) – Whether or not the Writer must open the file in binary mode for writing with Python 2. Defaults to `False`.
- **subsets** (*dict (str to list of str)*) – 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. Defaults to `None`.

- **logger** (*logging.Logger*) – A logger instance to use to log messages instead of creating a new one by default. Defaults to `None`.

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

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

Parameters

- **path** (*str*) – A path to the feature file we would like to create. The suffix to this filename must be `.arff`, `.csv`, `.jsonlines`, `.libsvm`, `.megam`, `.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.FeatureSet`) – The `FeatureSet` instance to dump to the output file.
- **kwargs** (*dict*) – 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.writers.Writer*

write()

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

From `experiments` Module

`skll.`**run_configuration**(*config_file*, *local=False*, *overwrite=True*, *queue=u'all.q'*, *hosts=None*, *write_summary=True*, *quiet=False*, *ablation=0*, *resume=False*, *log_level=20*)

Takes a configuration file and runs the specified jobs on the grid.

Parameters

- **config_file** (*str*) – Path to the configuration file we would like to use.
- **local** (*bool, optional*) – Should this be run locally instead of on the cluster? Defaults to `False`.
- **overwrite** (*bool, optional*) – If the model files already exist, should we overwrite them instead of re-using them? Defaults to `True`.
- **queue** (*str, optional*) – The DRMAA queue to use if we're running on the cluster. Defaults to `'all.q'`.

- **`hosts`** (*`list of str, optional`*) – If running on the cluster, these are the machines we should use. Defaults to `None`.
- **`write_summary`** (*`bool, optional`*) – Write a TSV file with a summary of the results. Defaults to `True`.
- **`quiet`** (*`bool, optional`*) – Suppress printing of "Loading…" messages. Defaults to `False`.
- **`ablation`** (*`int, optional`*) – Number of features to remove when doing an ablation experiment. If positive, we will perform repeated ablation runs for all combinations of features removing the specified number at a time. If `None`, we will use all combinations of all lengths. If 0, the default, no ablation is performed. If negative, a `ValueError` is raised. Defaults to 0.
- **`resume`** (*`bool, optional`*) – 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. Defaults to `False`.
- **`log_level`** (*`str, optional`*) – The level for logging messages. Defaults to `logging.INFO`.

Returns **result_json_paths** – A list of paths to .json results files for each variation in the experiment.

Return type list of str

Raises

- `ValueError` – If value for `"ablation"` is not a positive int or `None`.
- `OSError` – If the lenth of the `FeatureSet` name > 210.

From `learner` Module

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

Bases: `object`

A simpler learner interface around many scikit-learn classification and regression functions.

Parameters

- **`model_type`** (*`str`*) – Name of estimator to create (e.g., `'LogisticRegression'`). See the skll package documentation for valid options.

- **probability** (*bool, optional*) – Should learner return probabilities of all labels (instead of just label with highest probability)? Defaults to `False`.
- **pipeline** (*bool, optional*) – 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. Defaults to `False`.
- **feature_scaling** (*str, optional*) – 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 Defaults to 'none'.
- **model_kwargs** (*dict, optional*) – A dictionary of keyword arguments to pass to the initializer for the specified model. Defaults to `None`.
- **pos_label_str** (*str, optional*) – The string for the positive label in the binary classification setting. Otherwise, an arbitrary label is picked. Defaults to `None`.
- **min_feature_count** (*int, optional*) – The minimum number of examples a feature must have a nonzero value in to be included. Defaults to 1.
- **sampler** (*str, optional*) – The sampler to use for kernel approximation, if desired. Valid values are - 'AdditiveChi2Sampler' - 'Nystroem' - 'RBFSampler' - 'SkewedChi2Sampler' Defaults to `None`.
- **sampler_kwargs** (*dict, optional*) – A dictionary of keyword arguments to pass to the initializer for the specified sampler. Defaults to `None`.
- **custom_learner_path** (*str, optional*) – Path to module where a custom classifier is defined. Defaults to `None`.
- **logger** (*logging object, optional*) – A logging object. If `None` is passed, get logger from `__name__`. Defaults to `None`.

cross_validate(*examples, stratified=True, cv_folds=10, grid_search=True, grid_search_folds=3, grid_jobs=None, grid_objective=None, output_metrics=[], prediction_prefix=None, param_grid=None, shuffle=False, save_cv_folds=False, use_custom_folds_for_grid_search=True*)

Cross-validates a given model on the training examples.

Parameters

- **examples** (`skll.FeatureSet`) – The `FeatureSet` instance to cross-validate learner performance on.
- **stratified** (*bool, optional*) – Should we stratify the folds to ensure an even distribution of labels for each fold? Defaults to `True`.
- **cv_folds** (*int, optional*) – The number of folds to use for cross-validation, or a mapping from example IDs to folds. Defaults to 10.
- **grid_search** (*bool, optional*) – Should we do grid search when training each fold? Note: This will make this take *much* longer. Defaults to `False`.
- **grid_search_folds** (*int or dict, optional*) – The number of folds to use when doing the grid search, or a mapping from example IDs to folds. Defaults to 3.
- **grid_jobs** (*int, optional*) – 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. Defaults to `None`.
- **grid_objective** (*str, optional*) – The name of the objective function to use when doing the grid search. Must be specified if `grid_search` is `True`. Defaults to `None`.
- **output_metrics** (*list of str, optional*) – List of additional metric names to compute in addition to the metric used for grid search. Empty by default. Defaults to an empty list.
- **prediction_prefix** (*str, optional*) – If saving the predictions, this is the prefix that will be used for the filename. It will be followed by `"_predictions.tsv"` Defaults to `None`.
- **param_grid** (*list of dicts, optional*) – The parameter grid to traverse. Defaults to `None`.
- **shuffle** (*bool, optional*) – Shuffle examples before splitting into folds for CV. Defaults to `False`.
- **save_cv_folds** (*bool, optional*) – Whether to save the cv fold ids or not? Defaults to `False`.
- **use_custom_folds_for_grid_search** (*bool, optional*) – 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? Defaults to `True`.

Returns

- **results** (*list of 6-tuples*) – The confusion matrix, overall accuracy, per-label PRFs, model parameters, objective function score, and evaluation metrics (if any) for each fold.
- **grid_search_scores** (*list of floats*) – The grid search scores for each fold.
- **grid_search_cv_results_dicts** (*list of dicts*) – A 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.
- **skll_fold_ids** (*dict*) – A dictionary containing the test-fold number for each id if `save_cv_folds` is `True`, otherwise `None`.

Raises `ValueError` – If labels are not encoded as strings.

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

Evaluates a given model on a given dev or test `FeatureSet`.

Parameters

- **`examples`** (`skll.FeatureSet`) – The `FeatureSet` instance to evaluate the performance of the model on.
- **`prediction_prefix`** (*`str, optional`*) – If saving the predictions, this is the prefix that will be used for the filename. It will be followed by `"_predictions.tsv"` Defaults to `None`.
- **`append`** (*`bool, optional`*) – Should we append the current predictions to the file if it exists? Defaults to `False`.
- **`grid_objective`** (*`function, optional`*) – The objective function that was used when doing the grid search. Defaults to `None`.
- **`output_metrics`** (*`list of str, optional`*) – List of additional metric names to compute in addition to grid objective. Empty by default. Defaults to an empty list.

Returns **res** – 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 6-tuple

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

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

Parameters

- **`learner_path`** (*`str`*) – The path to a saved `Learner` instance file.

- **`logger`** (*`logging object, optional`*) – A logging object. If `None` is passed, get logger from `__name__`. Defaults to `None`.

Returns **learner** – The `Learner` instance loaded from the file.

Return type *sklI.Learner*

Raises

- `ValueError` – If the pickled object is not a `Learner` instance.
- `ValueError` – If the pickled version of the `Learner` instance is out of date.

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

Generates learning curves for a given model 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.FeatureSet`) – The `FeatureSet` instance to generate the learning curve on.
- **`cv_folds`** (*`int, optional`*) – The number of folds to use for cross-validation, or a mapping from example IDs to folds. Defaults to 10.
- **`metric`** (*`str`*) – The name of the metric function to use when computing the train and test scores for the learning curve.
- **`train_sizes`** (*`list of float or int, optional`*) – 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. Defaults to `np.linspace(0.1, 1.0, 5)`.

Returns

- **train_scores** (*list of float*) – The scores for the training set.
- **test_scores** (*list of float*) – The scores on the test set.
- **num_examples** (*list of int*) – The numbers of training examples used to generate the curve

`load`(*learner_path*)

Replace the current learner instance with a saved learner.

Parameters **`learner_path`** (*`str`*) – The path to a saved learner object file to load.

`model`
The underlying scikit-learn model

`model_kwargs`
A dictionary of the underlying scikit-learn model's keyword arguments

`model_params`
Model parameters (i.e., weights) for a `LinearModel` (e.g., `Ridge`) regression and liblinear models.

Returns

- **res** (*dict*) – A dictionary of labeled weights.
- **intercept** (*dict*) – A dictionary of intercept(s).

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

`model_type`
The model type (i.e., the class)

`predict`(*examples*, *prediction_prefix=None*, *append=False*, *class_labels=False*)
Uses a given model to generate predictions on a given `FeatureSet`.

Parameters

- **`examples`** (`skll.FeatureSet`) – The `FeatureSet` instance to predict labels for.
- **`prediction_prefix`** (*`str, optional`*) – If saving the predictions, this is the prefix that will be used for the filename. It will be followed by `"_predictions.tsv"` Defaults to `None`.
- **`append`** (*`bool, optional`*) – Should we append the current predictions to the file if it exists? Defaults to `False`.
- **`class_labels`** (*`bool, optional`*) – For classifier, should we convert class indices to their (str) labels? Defaults to `False`.

Returns **yhat** – The predictions returned by the `Learner` instance.

Return type array-like

Raises `MemoryError` – If process runs out of memory when converting to dense.

`probability`
Should 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** (*str*) – The path to save the `Learner` instance to.

train(*examples*, *param_grid=None*, *grid_search_folds=3*, *grid_search=True*, *grid_objective=None*, *grid_jobs=None*, *shuffle=False*, *create_label_dict=True*)

Train a classification model and return the model, score, feature vectorizer, scaler, label dictionary, and inverse label dictionary.

Parameters

- **examples** (`skll.FeatureSet`) – The `FeatureSet` instance to use for training.
- **param_grid** (*list of dicts, optional*) – The parameter grid to search through for grid search. If `None`, a default parameter grid will be used. Defaults to `None`.
- **grid_search_folds** (*int or dict, optional*) – The number of folds to use when doing the grid search, or a mapping from example IDs to folds. Defaults to 3.
- **grid_search** (*bool, optional*) – Should we do grid search? Defaults to `True`.
- **grid_objective** (*str, optional*) – The name of the objective function to use when doing the grid search. Must be specified if `grid_search` is `True`. Defaults to `None`.
- **grid_jobs** (*int, optional*) – 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. Defaults to `None`.
- **shuffle** (*bool, optional*) – Shuffle examples (e.g., for grid search CV.) Defaults to `False`.
- **create_label_dict** (*bool, optional*) – Should we create the label dictionary? This dictionary is used to map between string labels and their corresponding numerical values. This should only be done once per experiment, so when `cross_validate` calls `train`, `create_label_dict` gets set to `False`. Defaults to `True`.

Returns **tuple** – 1) The best grid search objective function score, or 0 if we're not doing grid search, and 2) a 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.

Return type (float, dict)

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.

From `metrics` Module

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

Calculate the F1 score of the least frequent label/class in `y_true` for `y_pred`.

Parameters

- **`y_true`** (*`array-like of float`*) – The true/actual/gold labels for the data.
- **`y_pred`** (*`array-like of float`*) – The predicted/observed labels for the data.

Returns **ret_score** – F1 score of the least frequent label.

Return type float

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

Calculates the kappa inter-rater agreement between two 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`** (*`array-like of float`*) – The true/actual/gold labels for the data.
- **`y_pred`** (*`array-like of float`*) – The predicted/observed labels for the data.

- **weights** (*str or np.array, optional*) – Specifies the weight matrix for the calculation. Options are

  ```
  -  None = unweighted-kappa
  -  'quadratic' = quadratic-weighted kappa
  -  'linear' = linear-weighted kappa
  -  two-dimensional numpy array = a custom matrix of
  ```

  weights. Each weight corresponds to the w_{ij} values in the wikipedia description of how to calculate weighted Cohen's kappa. Defaults to None.

- **allow_off_by_one** (*bool, optional*) – 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. Defaults to False.

Returns **k** – The kappa score, or weighted 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.

`skll.`**`kendall_tau`**(*y_true*, *y_pred*)

Calculate Kendall's tau between `y_true` and `y_pred`.

Parameters

- **y_true** (*array-like of float*) – The true/actual/gold labels for the data.
- **y_pred** (*array-like of float*) – The predicted/observed labels for the data.

Returns **ret_score** – Kendall's tau if well-defined, else 0.0

Return type float

`skll.`**`spearman`**(*y_true*, *y_pred*)

Calculate Spearman's rank correlation coefficient between `y_true` and `y_pred`.

Parameters

- **y_true** (*array-like of float*) – The true/actual/gold labels for the data.

- **y_pred** (*array-like of float*) – The predicted/observed labels for the data.

Returns ret_score – Spearman's rank correlation coefficient if well-defined, else 0.0

Return type float

`skll.pearson(y_true, y_pred)`

Calculate Pearson product-moment correlation coefficient between `y_true` and `y_pred`.

Parameters

- **y_true** (*array-like of float*) – The true/actual/gold labels for the data.
- **y_pred** (*array-like of float*) – The predicted/observed labels for the data.

Returns ret_score – Pearson product-moment correlation coefficient if well-defined, else 0.0

Return type float

1.6.3 `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`

Encapsulation of all of the features, values, and metadata about a given set of data. This replaces *ExamplesTuple* from older versions of SKLL.

Parameters

- **name** (*str*) – The name of this feature set.
- **ids** (*np.array*) – Example IDs for this set.
- **labels** (*np.array, optional*) – labels for this set. Defaults to `None`.

- **feature** (*list of dict or array-like, optional*) – The features for each instance represented as either a list of dictionaries or an array-like (if *vectorizer* is also specified). Defaults to `None`.
- **vectorizer** (*DictVectorizer or FeatureHasher, optional*) – Vectorizer which will be used to generate the feature matrix. Defaults to `None`.

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*)

Removes or keeps features and/or examples from the *Featureset* depending on the parameters. Filtering is done in-place.

Parameters

- **ids** (*list of str/float, optional*) – Examples to keep in the FeatureSet. If *None*, no ID filtering takes place. Defaults to `None`.
- **labels** (*list of str/float, optional*) – Labels that we want to retain examples for. If *None*, no label filtering takes place. Defaults to `None`.
- **features** (*list of str, optional*) – 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. Defaults to `None`.
- **inverse** (*bool, optional*) – Instead of keeping features and/or examples in lists, remove them. Defaults to `False`.

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

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

A version of ***__iter__*** that retains only the specified features and/or examples from the output.

Parameters

- **`ids`** (*`list of str/float, optional`*) – Examples to keep in the `FeatureSet`. If `None`, no ID filtering takes place. Defaults to `None`.
- **`labels`** (*`list of str/float, optional`*) – Labels that we want to retain examples for. If `None`, no label filtering takes place. Defaults to `None`.
- **`features`** (*`list of str, optional`*) – 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. Defaults to `None`.
- **`inverse`** (*`bool, optional`*) – Instead of keeping features and/or examples in lists, remove them. Defaults to `False`.

Yields

- **id_** (*str*) – The ID of the example.
- **label_** (*str*) – The label of the example.
- **feat_dict** (*dict*) – The feature dictionary, with feature name as the key and example value as the value.

Raises `ValueError` – If the vectorizer is not a *DictVectorizer*.

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

Helper function to create a `FeatureSet` instance from a *pandas.DataFrame*. 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`** (*`pd.DataFrame`*) – The pandas.DataFrame object to use as a `FeatureSet`.
- **`name`** (*`str`*) – The name of the output `FeatureSet` instance.
- **`labels_column`** (*`str, optional`*) – The name of the column containing the labels (data to predict). Defaults to `None`.
- **`vectorizer`** (*`DictVectorizer or FeatureHasher, optional`*) – Vectorizer which will be used to generate the feature matrix. Defaults to `None`.

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

Return type *skll.FeatureSet*

`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_by_ids` (*fs*, *ids_for_split1*, *ids_for_split2=None*)

Split the `FeatureSet` into two new `FeatureSet` instances based on the given IDs for the two splits.

Parameters

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

Returns

- **fs1** (*skll.FeatureSet*) – The first `FeatureSet`.
- **fs2** (*skll.FeatureSet*) – The second `FeatureSet`.

`data.readers` Module

This module handles loading data from various types of data files. A base `Reader` class is provided that is sub-classed for each data file type that is supported, e.g. `CSVReader`.

Notes about Label Conversion

All `Reader` sub-classes use the `safe_float` function internally to read in 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`. It should be noted that, if classification is being done with a feature set that is read in with one of the `Reader` sub-classes, care must be taken to ensure that labels do not get converted in unexpected ways. For example, classification labels should not be a mixture of `int`-converting and `float`-converting labels. Consider the situation below:

```
>>> 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. Be aware that it may be best to make use of the `class_map` keyword argument in such cases to map original labels to labels that convert only to `str`.

author Dan Blanchard (dblanchard@ets.org)

author Michael Heilman (mheilman@ets.org)

author Nitin Madnani (nmadnani@ets.org)

organization ETS

class `skll.data.readers.`**`ARFFReader`**(*path_or_list*, *pandas_kwargs=None*, ***kwargs*)

Bases: *`skll.data.readers.Reader`*

Reader for creating 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** (*str*) – The path to a comma-delimited file.
- **pandas_kwargs** (*dict or None, optional*) – Arguments that will be passed directly to the *pandas* I/O reader. Defaults to None.
- **kwargs** (*dict, optional*) – Other arguments to the Reader object.

`split_with_quotes`(*s*, *delimiter=u' '*, *header=None*, *quote_char=u"'"*, *escape_char=u'\\'*)

A replacement for string.split that won't split delimiters enclosed in quotes.

Parameters

- **s** (*str*) – The string with quotes to split
- **header** (*list or None, optional*) – The names of the header columns or None. Defaults to `None`.
- **delimiter** (*str, optional*) – The delimiter to split on. Defaults to `' '`.

- **quote_char** (*str, optional*) – The quote character to ignore. Defaults to `"'"`.
- **escape_char** (*str, optional*) – The escape character. Defaults to `'\'`.

class `skll.data.readers.`**CSVReader**(*path_or_list*, *pandas_kwargs=None*, ***kwargs*)

Bases: *skll.data.readers.Reader*

Reader for creating 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** (*str*) – The path to a comma-delimited file.
- **pandas_kwargs** (*dict or None, optional*) – Arguments that will be passed directly to the *pandas* I/O reader. Defaults to None.
- **kwargs** (*dict, optional*) – Other arguments to the Reader object.

class `skll.data.readers.`**DictListReader**(*path_or_list*, *pandas_kwargs=None*, ***kwargs*)

Bases: *skll.data.readers.Reader*

This class is to facilitate programmatic use of `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** (*str*) – The path to a comma-delimited file.
- **pandas_kwargs** (*dict or None, optional*) – Arguments that will be passed directly to the *pandas* data frame constructor. Defaults to None.
- **kwargs** (*dict, optional*) – Other arguments to the Reader object.

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

Bases: *skll.data.readers.Reader*

Reader to 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
```

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

Bases: `skll.data.readers.Reader`

Reader to create a `FeatureSet` instance from a MegaM -fvals file.

If example/instance IDs are included in the files, they must be specified as a comment line directly preceding the line with feature values.

class skll.data.readers.**NDJReader**(*path_or_list*, *pandas_kwargs=None*, ***kwargs*)

Bases: `skll.data.readers.Reader`

Reader to 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** (*str*) – The path to a comma-delimited file.
- **pandas_kwargs** (*dict or None, optional*) – Arguments that will be passed directly to the *pandas* I/O reader. Defaults to None.
- **kwargs** (*dict, optional*) – Other arguments to the Reader object.

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

Bases: `object`

A helper class to make picklable iterators out of example dictionary generators.

Parameters

- **path_or_list** (*str or list of dict*) – Path or a list of example dictionaries.
- **quiet** (*bool, optional*) – Do not print "Loading…" status message to stderr. Defaults to `True`.

- **ids_to_floats** (*bool, optional*) – Convert IDs to float to save memory. Will raise error if we encounter an a non-numeric ID. Defaults to `False`.
- **label_col** (*str, optional*) – 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. Defaults to `'y'`.
- **id_col** (*str, optional*) – 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. Defaults to `'id'`.
- **class_map** (*dict, optional*) – 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. Defaults to `None`.
- **sparse** (*bool, optional*) – Whether or not to store the features in a numpy CSR matrix when using a DictVectorizer to vectorize the features. Defaults to `True`.
- **feature_hasher** (*bool, optional*) – Whether or not a FeatureHasher should be used to vectorize the features. Defaults to `False`.
- **num_features** (*int, optional*) – 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. Defaults to `None`.
- **logger** (*logging.Logger, optional*) – A logger instance to use to log messages instead of creating a new one by default. Defaults to `None`.

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

Instantiate the appropriate Reader sub-class based on the file extension of the given path. Or use a dictionary reader if the input is a list of dictionaries.

Parameters

- **path_or_list** (*str or list of dicts*) – A path or list of example dictionaries.
- **kwargs** (*dict, optional*) – 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.Reader*

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

read()

Loads examples in the *.arff*, *.csv*, *.jsonlines*, *.libsvm*, *.megam*, *.ndj*, or *.tsv* formats.

Returns **feature_set** – `FeatureSet` instance representing the input file.

Return type *skll.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 `skll.data.readers.`**TSVReader**(*path_or_list*, *pandas_kwargs=None*, ***kwargs*)

Bases: *skll.data.readers.CSVReader*

Reader for creating 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.
- **pandas_kwargs** (*dict or None, optional*) – Arguments that will be passed directly to the *pandas* I/O reader. Defaults to None.
- **kwargs** (*dict, optional*) – Other arguments to the Reader object.

`skll.data.readers.`**safe_float**(*text*, *replace_dict=None*, *logger=None*)

Attempts to convert a string to an int, and then a float, but if neither is possible, returns the original string value.

Parameters

- **text** (*str*) – The text to convert.
- **replace_dict** (*dict, optional*) – Mapping from text to replacement text values. This is mainly used for collapsing multiple labels into a single class. Replacing happens before conversion to floats. Anything not in the mapping will be kept the same. Defaults to `None`.
- **logger** (*logging.Logger*) – The Logger instance to use to log messages. Used instead of creating a new Logger instance by default. Defaults to `None`.

Returns **text** – The text value converted to int or float, if possible

Return type int or float or str

`data.writers` Module

Handles loading data from various types of data files.

author Dan Blanchard (dblanchard@ets.org)

author Michael Heilman (mheilman@ets.org)

author Nitin Madnani (nmadnani@ets.org)

organization ETS

class `sklll.data.writers.`**`ARFFWriter`**(*path*, *feature_set*, *pandas_kwargs=None*, ***kwargs*)

Bases: *`skll.data.writers.Writer`*

Writer for writing out FeatureSets as ARFF files.

Parameters

- **`path`** (*`str`*) – 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.FeatureSet`) – The `FeatureSet` instance to dump to the output file.
- **`relation`** (*`str, optional`*) – The name of the relation in the ARFF file. Defaults to `'skll_relation'`.
- **`regression`** (*`bool, optional`*) – Is this an ARFF file to be used for regression? Defaults to `False`.
- **`pandas_kwargs`** (*`dict or None, optional`*) – Arguments that will be passed directly to the *pandas* I/O reader. Defaults to None.
- **`kwargs`** (*`dict, optional`*) – The arguments to the `Writer` object being instantiated.

class `skll.data.writers.`**`CSVWriter`**(*path*, *feature_set*, *pandas_kwargs=None*, ***kwargs*)

Bases: *`skll.data.writers.Writer`*

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

Parameters

- **`path`** (*`str`*) – 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.FeatureSet`) – The `FeatureSet` instance to dump to the output file.
- **pandas_kwargs** (*dict or None, optional*) – Arguments that will be passed directly to the *pandas* I/O reader. Defaults to None.
- **kwargs** (*dict, optional*) – The arguments to the `Writer` object being instantiated.

class `skll.data.writers.`**LibSVMWriter**(*path*, *feature_set*, ***kwargs*)

Bases: *skll.data.writers.Writer*

Writer for writing out FeatureSets as LibSVM/SVMLight files.

Parameters

- **path** (*str*) – 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.FeatureSet`) – The `FeatureSet` instance to dump to the output file.
- **kwargs** (*dict, optional*) – The arguments to the `Writer` object being instantiated.

class `skll.data.writers.`**MegaMWriter**(*path*, *feature_set*, ***kwargs*)

Bases: *skll.data.writers.Writer*

Writer for writing out FeatureSets as MegaM files.

class `skll.data.writers.`**NDJWriter**(*path*, *feature_set*, *pandas_kwargs=None*, ***kwargs*)

Bases: *skll.data.writers.Writer*

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

Parameters

- **path** (*str*) – 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.FeatureSet`) – The `FeatureSet` instance to dump to the output file.
- **pandas_kwargs** (*dict or None, optional*) – Arguments that will be passed directly to the *pandas* I/O reader. Defaults to None.
- **kwargs** (*dict, optional*) – The arguments to the `Writer` object being instantiated.

class `skll.data.writers.`**`TSVWriter`**(*path*, *feature_set*, *pandas_kwargs=None*, ***kwargs*)

Bases: *`skll.data.writers.CSVWriter`*

Writer for writing out FeatureSets as TSV files.

Parameters

- **`path`** (*`str`*) – 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.FeatureSet`) – The `FeatureSet` instance to dump to the output file.
- **`pandas_kwargs`** (*`dict or None, optional`*) – Arguments that will be passed directly to the *pandas* I/O reader. Defaults to None.
- **`kwargs`** (*`dict, optional`*) – The arguments to the `Writer` object being instantiated.

class `skll.data.writers.`**`Writer`**(*path*, *feature_set*, ***kwargs*)

Bases: `object`

Helper class for writing out FeatureSets to files on disk.

Parameters

- **`path`** (*`str`*) – A path to the feature file we would like to create. The suffix to this filename must be `.arff`, `.csv`, `.jsonlines`, `.libsvm`, `.megam`, `.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.FeatureSet`) – The `FeatureSet` instance to dump to the file.
- **`quiet`** (*`bool`*) – Do not print "Writing…" status message to stderr. Defaults to `True`.
- **`requires_binary`** (*`bool`*) – Whether or not the Writer must open the file in binary mode for writing with Python 2. Defaults to `False`.
- **`subsets`** (*`dict (str to list of str)`*) – 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. Defaults to `None`.

- **logger** (*logging.Logger*) – A logger instance to use to log messages instead of creating a new one by default. Defaults to `None`.

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

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

Parameters

- **path** (*str*) – A path to the feature file we would like to create. The suffix to this filename must be `.arff`, `.csv`, `.jsonlines`, `.libsvm`, `.megam`, `.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.FeatureSet`) – The `FeatureSet` instance to dump to the output file.
- **kwargs** (*dict*) – 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.writers.Writer*

write()

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

1.6.4 `experiments` Module

Functions related to running experiments and parsing configuration files.

author Dan Blanchard (dblanchard@ets.org)

author Michael Heilman (mheilman@ets.org)

author Nitin Madnani (nmadnani@ets.org)

author Chee Wee Leong (cleong@ets.org)

class sklearn.experiments.**NumpyTypeEncoder**(*skipkeys=False*, *ensure_ascii=True*, *check_circular=True*, *allow_nan=True*, *sort_keys=False*, *indent=None*, *separators=None*, *encoding='utf-8'*, *default=None*)

Bases: `json.encoder.JSONEncoder`

This class is used when serializing results, particularly the input label values if the input has int-valued labels. Numpy int64 objects can't be serialized by the json module, so we must convert them to int objects.

A related issue where this was adapted from: https://stackoverflow.com/questions/11561932/why-does-json-dumpslistnp-arange5-fail-while-json-dumpsnp-arange5-tolis

default(*obj*)

Implement this method in a subclass such that it returns a serializable object for `o`, or calls the base implementation (to raise a `TypeError`).

For example, to support arbitrary iterators, you could implement default like this:

```
def default(self, o):
    try:
        iterable = iter(o)
    except TypeError:
        pass
    else:
        return list(iterable)
    # Let the base class default method raise the TypeError
    return JSONEncoder.default(self, o)
```

skll.experiments.**run_configuration**(*config_file*, *local=False*, *overwrite=True*, *queue=u'all.q'*, *hosts=None*, *write_summary=True*, *quiet=False*, *ablation=0*, *resume=False*, *log_level=20*)

Takes a configuration file and runs the specified jobs on the grid.

Parameters

- **config_file** (*str*) – Path to the configuration file we would like to use.
- **local** (*bool, optional*) – Should this be run locally instead of on the cluster? Defaults to `False`.
- **overwrite** (*bool, optional*) – If the model files already exist, should we overwrite them instead of re-using them? Defaults to `True`.

- **queue** (*str, optional*) – The DRMAA queue to use if we're running on the cluster. Defaults to `'all.q'`.
- **hosts** (*list of str, optional*) – If running on the cluster, these are the machines we should use. Defaults to `None`.
- **write_summary** (*bool, optional*) – Write a TSV file with a summary of the results. Defaults to `True`.
- **quiet** (*bool, optional*) – Suppress printing of "Loading…" messages. Defaults to `False`.
- **ablation** (*int, optional*) – Number of features to remove when doing an ablation experiment. If positive, we will perform repeated ablation runs for all combinations of features removing the specified number at a time. If `None`, we will use all combinations of all lengths. If 0, the default, no ablation is performed. If negative, a `ValueError` is raised. Defaults to 0.
- **resume** (*bool, optional*) – 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. Defaults to `False`.
- **log_level** (*str, optional*) – The level for logging messages. Defaults to `logging.INFO`.

Returns **result_json_paths** – A list of paths to .json results files for each variation in the experiment.

Return type list of str

Raises

- `ValueError` – If value for `"ablation"` is not a positive int or `None`.
- `OSError` – If the lenth of the `FeatureSet` name > 210.

1.6.5 `learner` Module

Provides easy-to-use wrapper around scikit-learn.

author Michael Heilman (mheilman@ets.org)

author Nitin Madnani (nmadnani@ets.org)

author Dan Blanchard (dblanchard@ets.org)

author Aoife Cahill (acahill@ets.org)

organization ETS

class `sklearn.learner.`**`Densifier`**

Bases: `sklearn.base.BaseEstimator`, `sklearn.base.TransformerMixin`

A custom pipeline stage that will be inserted into the learner pipeline attribute to accommodate the situation when SKLL needs to manually convert feature arrays from sparse to dense. For example, when features are being hashed but we are also doing centering using the feature means.

`fit_transform`(*X*, *y=None*)

Fit to data, then transform it.

Fits transformer to X and y with optional parameters fit_params and returns a transformed version of X.

Parameters

- **X** (*numpy array of shape [n_samples, n_features]*) – Training set.
- **y** (*numpy array of shape [n_samples]*) – Target values.

Returns **X_new** – Transformed array.

Return type numpy array of shape [n_samples, n_features_new]

class `skll.learner.`**`FilteredLeaveOneGroupOut`**(*keep*, *example_ids*)

Bases: `sklearn.model_selection._split.LeaveOneGroupOut`

Version of `LeaveOneGroupOut` cross-validation iterator that only outputs indices of instances with IDs in a prespecified set.

Parameters

- **keep** (*set of str*) – A set of IDs to keep.
- **example_ids** (*list of str, of length n_samples*) – A list of example IDs.

`split`(*X*, *y*, *groups*)

Generate indices to split data into training and test set.

Parameters

- **X** (*array-like, with shape (n_samples, n_features)*) – Training data, where n_samples is the number of samples and n_features is the number of features.
- **y** (*array-like, of length n_samples*) – The target variable for supervised learning problems.
- **groups** (*array-like, with shape (n_samples,)*) – Group labels for the samples used while splitting the dataset into train/test set.

Yields

- **train_index** (*np.array*) – The training set indices for that split.
- **test_index** (*np.array*) – The testing set indices for that split.

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

Bases: `object`

A simpler learner interface around many scikit-learn classification and regression functions.

Parameters

- **`model_type`** (*`str`*) – Name of estimator to create (e.g., `'LogisticRegression'`). See the skll package documentation for valid options.
- **`probability`** (*`bool, optional`*) – Should learner return probabilities of all labels (instead of just label with highest probability)? Defaults to `False`.
- **`pipeline`** (*`bool, optional`*) – 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. Defaults to `False`.
- **`feature_scaling`** (*`str, optional`*) – 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 Defaults to 'none'.
- **`model_kwargs`** (*`dict, optional`*) – A dictionary of keyword arguments to pass to the initializer for the specified model. Defaults to `None`.
- **`pos_label_str`** (*`str, optional`*) – The string for the positive label in the binary classification setting. Otherwise, an arbitrary label is picked. Defaults to `None`.
- **`min_feature_count`** (*`int, optional`*) – The minimum number of examples a feature must have a nonzero value in to be included. Defaults to 1.

- **sampler** (*str, optional*) – The sampler to use for kernel approximation, if desired. Valid values are - 'AdditiveChi2Sampler' - 'Nystroem' - 'RBFSampler' - 'SkewedChi2Sampler' Defaults to `None`.
- **sampler_kwargs** (*dict, optional*) – A dictionary of keyword arguments to pass to the initializer for the specified sampler. Defaults to `None`.
- **custom_learner_path** (*str, optional*) – Path to module where a custom classifier is defined. Defaults to `None`.
- **logger** (*logging object, optional*) – A logging object. If `None` is passed, get logger from `__name__`. Defaults to `None`.

cross_validate(*examples, stratified=True, cv_folds=10, grid_search=True, grid_search_folds=3, grid_jobs=None, grid_objective=None, output_metrics=[], prediction_prefix=None, param_grid=None, shuffle=False, save_cv_folds=False, use_custom_folds_for_grid_search=True*)

Cross-validates a given model on the training examples.

Parameters

- **examples** (`skll.FeatureSet`) – The `FeatureSet` instance to cross-validate learner performance on.
- **stratified** (*bool, optional*) – Should we stratify the folds to ensure an even distribution of labels for each fold? Defaults to `True`.
- **cv_folds** (*int, optional*) – The number of folds to use for cross-validation, or a mapping from example IDs to folds. Defaults to 10.
- **grid_search** (*bool, optional*) – Should we do grid search when training each fold? Note: This will make this take *much* longer. Defaults to `False`.
- **grid_search_folds** (*int or dict, optional*) – The number of folds to use when doing the grid search, or a mapping from example IDs to folds. Defaults to 3.
- **grid_jobs** (*int, optional*) – 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. Defaults to `None`.
- **grid_objective** (*str, optional*) – The name of the objective function to use when doing the grid search. Must be specified if `grid_search` is `True`. Defaults to `None`.
- **output_metrics** (*list of str, optional*) – List of additional metric names to compute in addition to the metric used for grid

search. Empty by default. Defaults to an empty list.

- **prediction_prefix** (*str, optional*) – If saving the predictions, this is the prefix that will be used for the filename. It will be followed by `"_predictions.tsv"` Defaults to `None`.
- **param_grid** (*list of dicts, optional*) – The parameter grid to traverse. Defaults to `None`.
- **shuffle** (*bool, optional*) – Shuffle examples before splitting into folds for CV. Defaults to `False`.
- **save_cv_folds** (*bool, optional*) – Whether to save the cv fold ids or not? Defaults to `False`.
- **use_custom_folds_for_grid_search** (*bool, optional*) – 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? Defaults to `True`.

Returns

- **results** (*list of 6-tuples*) – The confusion matrix, overall accuracy, per-label PRFs, model parameters, objective function score, and evaluation metrics (if any) for each fold.
- **grid_search_scores** (*list of floats*) – The grid search scores for each fold.
- **grid_search_cv_results_dicts** (*list of dicts*) – A 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.
- **skll_fold_ids** (*dict*) – A dictionary containing the test-fold number for each id if `save_cv_folds` is `True`, otherwise `None`.

Raises `ValueError` – If labels are not encoded as strings.

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

Evaluates a given model on a given dev or test `FeatureSet`.

Parameters

- **examples** (`skll.FeatureSet`) – The `FeatureSet` instance to evaluate the performance of the model on.
- **prediction_prefix** (*str, optional*) – If saving the predictions, this is the prefix that will be used for the filename. It will be followed by `"_predictions.tsv"` Defaults to `None`.

- **append** (*bool, optional*) – Should we append the current predictions to the file if it exists? Defaults to `False`.
- **grid_objective** (*function, optional*) – The objective function that was used when doing the grid search. Defaults to `None`.
- **output_metrics** (*list of str, optional*) – List of additional metric names to compute in addition to grid objective. Empty by default. Defaults to an empty list.

Returns **res** – 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 6-tuple

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

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

Parameters

- **learner_path** (*str*) – The path to a saved `Learner` instance file.
- **logger** (*logging object, optional*) – A logging object. If `None` is passed, get logger from `__name__`. Defaults to `None`.

Returns **learner** – The `Learner` instance loaded from the file.

Return type *skll.Learner*

Raises

- `ValueError` – If the pickled object is not a `Learner` instance.
- `ValueError` – If the pickled version of the `Learner` instance is out of date.

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

Generates learning curves for a given model 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.FeatureSet`) – The `FeatureSet` instance to generate the learning curve on.
- **cv_folds** (*int, optional*) – The number of folds to use for cross-validation, or a mapping from example IDs to folds. Defaults to 10.
- **metric** (*str*) – The name of the metric function to use when computing the train and test scores for the learning curve.

- **`train_sizes`** (*`list of float or int, optional`*) – 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. Defaults to `np.linspace(0.1, 1.0, 5)`.

Returns

- **train_scores** (*list of float*) – The scores for the training set.
- **test_scores** (*list of float*) – The scores on the test set.
- **num_examples** (*list of int*) – The numbers of training examples used to generate the curve

`load`(*learner_path*)
Replace the current learner instance with a saved learner.

Parameters **`learner_path`** (*`str`*) – The path to a saved learner object file to load.

`model`
The underlying scikit-learn model

`model_kwargs`
A dictionary of the underlying scikit-learn model's keyword arguments

`model_params`
Model parameters (i.e., weights) for a `LinearModel` (e.g., `Ridge`) regression and liblinear models.

Returns

- **res** (*dict*) – A dictionary of labeled weights.
- **intercept** (*dict*) – A dictionary of intercept(s).

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

`model_type`
The model type (i.e., the class)

`predict`(*examples*, *prediction_prefix=None*, *append=False*, *class_labels=False*)
Uses a given model to generate predictions on a given `FeatureSet`.

Parameters

- **`examples`** (`skll.FeatureSet`) – The `FeatureSet` instance to predict labels for.

- **prediction_prefix** (*str, optional*) – If saving the predictions, this is the prefix that will be used for the filename. It will be followed by `"_predictions.tsv"` Defaults to `None`.
- **append** (*bool, optional*) – Should we append the current predictions to the file if it exists? Defaults to `False`.
- **class_labels** (*bool, optional*) – For classifier, should we convert class indices to their (str) labels? Defaults to `False`.

Returns **yhat** – The predictions returned by the `Learner` instance.

Return type array-like

Raises `MemoryError` – If process runs out of memory when converting to dense.

probability

Should 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** (*str*) – The path to save the `Learner` instance to.

train (*examples, param_grid=None, grid_search_folds=3, grid_search=True, grid_objective=None, grid_jobs=None, shuffle=False, create_label_dict=True*)

Train a classification model and return the model, score, feature vectorizer, scaler, label dictionary, and inverse label dictionary.

Parameters

- **examples** (`skll.FeatureSet`) – The `FeatureSet` instance to use for training.
- **param_grid** (*list of dicts, optional*) – The parameter grid to search through for grid search. If `None`, a default parameter grid will be used. Defaults to `None`.
- **grid_search_folds** (*int or dict, optional*) – The number of folds to use when doing the grid search, or a mapping from example IDs to folds. Defaults to 3.
- **grid_search** (*bool, optional*) – Should we do grid search? Defaults to `True`.
- **grid_objective** (*str, optional*) – The name of the objective function to use when doing the grid search. Must be specified if `grid_search` is `True`. Defaults to `None`.

- **`grid_jobs`** (*`int, optional`*) – 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. Defaults to `None`.
- **`shuffle`** (*`bool, optional`*) – Shuffle examples (e.g., for grid search CV.) Defaults to `False`.
- **`create_label_dict`** (*`bool, optional`*) – Should we create the label dictionary? This dictionary is used to map between string labels and their corresponding numerical values. This should only be done once per experiment, so when `cross_validate` calls `train`, `create_label_dict` gets set to `False`. Defaults to `True`.

Returns **tuple** – 1) The best grid search objective function score, or 0 if we're not doing grid search, and 2) a 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.

Return type (float, dict)

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.

class `skll.learner.`**`RescaledAdaBoostRegressor`**(*constrain=True*, *rescale=True*, ***kwargs*)

Bases: `sklearn.ensemble.weight_boosting.AdaBoostRegressor`

`fit`(*X*, *y=None*)

Build a boosted regressor from the training set (X, y).

Parameters

- **`X`** (*`{array-like, sparse matrix} of shape = [n_samples, n_features]`*) – The training input samples. Sparse matrix can be CSC, CSR, COO, DOK, or LIL. DOK and LIL are converted to CSR.
- **`y`** (*`array-like of shape = [n_samples]`*) – The target values (real numbers).
- **`sample_weight`** (*`array-like of shape = [n_samples], optional`*) – Sample weights. If None, the sample weights are initialized to 1 / n_samples.

Returns self

Return type object

predict(*X*)

Predict regression value for X.

The predicted regression value of an input sample is computed as the weighted median prediction of the classifiers in the ensemble.

Parameters **X** (*{array-like, sparse matrix} of shape = [n_samples, n_features]*) – The training input samples. Sparse matrix can be CSC, CSR, COO, DOK, or LIL. DOK and LIL are converted to CSR.

Returns **y** – The predicted regression values.

Return type array of shape = [n_samples]

class skll.learner.**RescaledBayesianRidge**(*constrain=True*, *rescale=True*, ***kwargs*)

Bases: `sklearn.linear_model.bayes.BayesianRidge`

fit(*X*, *y=None*)

Fit the model

Parameters

- **X** (*numpy array of shape [n_samples,n_features]*) – Training data
- **y** (*numpy array of shape [n_samples]*) – Target values. Will be cast to X's dtype if necessary
- **sample_weight** (*numpy array of shape [n_samples]*) – Individual weights for each sample

  New in version 0.20: parameter *sample_weight* support to BayesianRidge.

Returns self

Return type returns an instance of self.

predict(*X*)

Predict using the linear model.

In addition to the mean of the predictive distribution, also its standard deviation can be returned.

Parameters

- **X** (*{array-like, sparse matrix}, shape = (n_samples, n_features)*) – Samples.

- **return_std** (*boolean, optional*) – Whether to return the standard deviation of posterior prediction.

Returns

- **y_mean** (*array, shape = (n_samples,)*) – Mean of predictive distribution of query points.
- **y_std** (*array, shape = (n_samples,)*) – Standard deviation of predictive distribution of query points.

class `skll.learner.`**`RescaledDecisionTreeRegressor`**(*constrain=True*, *rescale=True*, ***kwargs*)

Bases: `sklearn.tree.tree.DecisionTreeRegressor`

`fit`(*X*, *y=None*)

Build a decision tree regressor from the training set (X, y).

Parameters

- **X** (*array-like or sparse matrix, shape = [n_samples, n_features]*) – The training input samples. Internally, it will be converted to `dtype=np.float32` and if a sparse matrix is provided to a sparse `csc_matrix`.
- **y** (*array-like, shape = [n_samples] or [n_samples, n_outputs]*) – The target values (real numbers). Use `dtype=np.float64` and `order='C'` for maximum efficiency.
- **sample_weight** (*array-like, shape = [n_samples] or None*) – Sample weights. If None, then samples are equally weighted. Splits that would create child nodes with net zero or negative weight are ignored while searching for a split in each node.
- **check_input** (*boolean, (default=True)*) – Allow to bypass several input checking. Don't use this parameter unless you know what you do.
- **X_idx_sorted** (*array-like, shape = [n_samples, n_features], optional*) – The indexes of the sorted training input samples. If many tree are grown on the same dataset, this allows the ordering to be cached between trees. If None, the data will be sorted here. Don't use this parameter unless you know what to do.

Returns self

Return type object

`predict`(*X*)

Predict class or regression value for X.

For a classification model, the predicted class for each sample in X is returned. For a regression model, the predicted value based on X is returned.

Parameters

- **X** (*array-like or sparse matrix of shape = [n_samples, n_features]*) – The input samples. Internally, it will be converted to `dtype=np.float32` and if a sparse matrix is provided to a sparse `csr_matrix`.
- **check_input** (*boolean, (default=True)*) – Allow to bypass several input checking. Don't use this parameter unless you know what you do.

Returns **y** – The predicted classes, or the predict values.

Return type array of shape = [n_samples] or [n_samples, n_outputs]

class `skll.learner.`**`RescaledElasticNet`**(*constrain=True*, *rescale=True*, ***kwargs*)

Bases: `sklearn.linear_model.coordinate_descent.ElasticNet`

`fit`(*X*, *y=None*)

Fit model with coordinate descent.

Parameters

- **X** (*ndarray or scipy.sparse matrix, (n_samples, n_features)*) – Data
- **y** (*ndarray, shape (n_samples,) or (n_samples, n_targets)*) – Target. Will be cast to X's dtype if necessary
- **check_input** (*boolean, (default=True)*) – Allow to bypass several input checking. Don't use this parameter unless you know what you do.

Notes

Coordinate descent is an algorithm that considers each column of data at a time hence it will automatically convert the X input as a Fortran-contiguous numpy array if necessary.

To avoid memory re-allocation it is advised to allocate the initial data in memory directly using that format.

`predict`(*X*)

Predict using the linear model

Parameters **X** (*array_like or sparse matrix, shape (n_samples, n_features)*) – Samples.

Returns **C** – Returns predicted values.

Return type array, shape (n_samples,)

class `skll.learner.`**`RescaledGradientBoostingRegressor`**(*constrain=True*, *rescale=True*, ***kwargs*)

Bases: `sklearn.ensemble.gradient_boosting.GradientBoostingRegressor`

`fit`(*X*, *y=None*)

Fit the gradient boosting model.

Parameters

- **X** (*{array-like, sparse matrix}, shape (n_samples, n_features)*) – The input samples. Internally, it will be converted to `dtype=np.float32` and if a sparse matrix is provided to a sparse `csr_matrix`.
- **y** (*array-like, shape (n_samples,)*) – Target values (strings or integers in classification, real numbers in regression) For classification, labels must correspond to classes.
- **sample_weight** (*array-like, shape (n_samples,) or None*) – Sample weights. If None, then samples are equally weighted. Splits that would create child nodes with net zero or negative weight are ignored while searching for a split in each node. In the case of classification, splits are also ignored if they would result in any single class carrying a negative weight in either child node.
- **monitor** (*callable, optional*) – The monitor is called after each iteration with the current iteration, a reference to the estimator and the local variables of `_fit_stages` as keyword arguments `callable(i, self, locals())`. If the callable returns `True` the fitting procedure is stopped. The monitor can be used for various things such as computing held-out estimates, early stopping, model introspect, and snapshoting.

Returns **self**

Return type object

`predict`(*X*)

Predict regression target for X.

Parameters **X** (*{array-like, sparse matrix}, shape (n_samples, n_features)*) – The input samples. Internally, it will be converted to `dtype=np.float32` and if a sparse matrix is provided to a sparse `csr_matrix`.

Returns **y** – The predicted values.

Return type array, shape (n_samples,)

class `sklL.learner.`**`RescaledHuberRegressor`**(*constrain=True*, *rescale=True*, ***kwargs*)

Bases: `sklearn.linear_model.huber.HuberRegressor`

`fit`(*X*, *y=None*)

Fit the model according to the given training data.

Parameters

- **X** (*array-like, shape (n_samples, n_features)*) – Training vector, where n_samples in the number of samples and n_features is the number of features.
- **y** (*array-like, shape (n_samples,)*) – Target vector relative to X.
- **sample_weight** (*array-like, shape (n_samples,)*) – Weight given to each sample.

Returns **self**

Return type object

`predict`(*X*)

Predict using the linear model

Parameters **X** (*array_like or sparse matrix, shape (n_samples, n_features)*) – Samples.

Returns **C** – Returns predicted values.

Return type array, shape (n_samples,)

class `skll.learner.`**`RescaledKNeighborsRegressor`**(*constrain=True*, *rescale=True*, ***kwargs*)

Bases: `sklearn.neighbors.regression.KNeighborsRegressor`

`fit`(*X*, *y=None*)

Fit the model using X as training data and y as target values

Parameters

- **X** (*{array-like, sparse matrix, BallTree, KDTree}*) – Training data. If array or matrix, shape [n_samples, n_features], or [n_samples, n_samples] if metric='precomputed'.
- **y** (*{array-like, sparse matrix}*) –

Target values, array of float values, shape = [n_samples] or [n_samples, n_outputs]

predict (*X*)

Predict the target for the provided data

Parameters **X** (*array-like, shape (n_query, n_features), or (n_query, n_indexed) if metric == 'precomputed'*) – Test samples.

Returns **y** – Target values

Return type array of int, shape = [n_samples] or [n_samples, n_outputs]

class sklll.learner.**RescaledLars** (*constrain=True*, *rescale=True*, ***kwargs*)

Bases: `sklearn.linear_model.least_angle.Lars`

fit (*X*, *y=None*)

Fit the model using X, y as training data.

Parameters

- **X** (*array-like, shape (n_samples, n_features)*) – Training data.
- **y** (*array-like, shape (n_samples,) or (n_samples, n_targets)*) – Target values.
- **Xy** (*array-like, shape (n_samples,) or (n_samples, n_targets), optional*) – Xy = np.dot(X.T, y) that can be precomputed. It is useful only when the Gram matrix is precomputed.

Returns **self** – returns an instance of self.

Return type object

predict (*X*)

Predict using the linear model

Parameters **X** (*array_like or sparse matrix, shape (n_samples, n_features)*) – Samples.

Returns **C** – Returns predicted values.

Return type array, shape (n_samples,)

class skll.learner.**RescaledLasso** (*constrain=True*, *rescale=True*, ***kwargs*)

Bases: `sklearn.linear_model.coordinate_descent.Lasso`

fit (*X*, *y=None*)

Fit model with coordinate descent.

Parameters

- **X** (*ndarray or scipy.sparse matrix, (n_samples, n_features)*) – Data
- **y** (*ndarray, shape (n_samples,) or (n_samples, n_targets)*) – Target. Will be cast to X's dtype if necessary
- **check_input** (*boolean, (default=True)*) – Allow to bypass several input checking. Don't use this parameter unless you know what you do.

Notes

Coordinate descent is an algorithm that considers each column of data at a time hence it will automatically convert the X input as a Fortran-contiguous numpy array if necessary.

To avoid memory re-allocation it is advised to allocate the initial data in memory directly using that format.

predict(*X*)

Predict using the linear model

Parameters X (*array_like or sparse matrix, shape (n_samples, n_features)*) – Samples.

Returns C – Returns predicted values.

Return type array, shape (n_samples,)

class skll.learner.**RescaledLinearRegression**(*constrain=True, rescale=True, **kwargs*)

Bases: `sklearn.linear_model.base.LinearRegression`

fit(*X, y=None*)

Fit linear model.

Parameters

- **X** (*array-like or sparse matrix, shape (n_samples, n_features)*) – Training data
- **y** (*array_like, shape (n_samples, n_targets)*) – Target values. Will be cast to X's dtype if necessary
- **sample_weight** (*numpy array of shape [n_samples]*) – Individual weights for each sample

  New in version 0.17: parameter *sample_weight* support to LinearRegression.

Returns self

Return type returns an instance of self.

predict(*X*)

Predict using the linear model

Parameters **X** (*array_like or sparse matrix, shape (n_samples, n_features)*) – Samples.

Returns **C** – Returns predicted values.

Return type array, shape (n_samples,)

class skll.learner.**RescaledLinearSVR**(*constrain=True*, *rescale=True*, ***kwargs*)

Bases: `sklearn.svm.classes.LinearSVR`

fit(*X*, *y=None*)

Fit the model according to the given training data.

Parameters

- **X** (*{array-like, sparse matrix}, shape = [n_samples, n_features]*) – Training vector, where n_samples in the number of samples and n_features is the number of features.
- **y** (*array-like, shape = [n_samples]*) – Target vector relative to X
- **sample_weight** (*array-like, shape = [n_samples], optional*) – Array of weights that are assigned to individual samples. If not provided, then each sample is given unit weight.

Returns self

Return type object

predict(*X*)

Predict using the linear model

Parameters **X** (*array_like or sparse matrix, shape (n_samples, n_features)*) – Samples.

Returns **C** – Returns predicted values.

Return type array, shape (n_samples,)

class skll.learner.**RescaledMLPRegressor**(*constrain=True*, *rescale=True*, ***kwargs*)

Bases: `sklearn.neural_network.multilayer_perceptron.MLPRegressor`

fit (*X*, *y=None*)

Fit the model to data matrix X and target(s) y.

Parameters

- **X** (*array-like or sparse matrix, shape (n_samples, n_features)*) – The input data.
- **y** (*array-like, shape (n_samples,) or (n_samples, n_outputs)*) – The target values (class labels in classification, real numbers in regression).

Returns self

Return type returns a trained MLP model.

predict (*X*)

Predict using the multi-layer perceptron model.

Parameters X (*{array-like, sparse matrix}, shape (n_samples, n_features)*) – The input data.

Returns y – The predicted values.

Return type array-like, shape (n_samples, n_outputs)

class `skll.learner.`**RescaledRANSACRegressor** (*constrain=True*, *rescale=True*, ***kwargs*)

Bases: `sklearn.linear_model.ransac.RANSACRegressor`

fit (*X*, *y=None*)

Fit estimator using RANSAC algorithm.

Parameters

- **X** (*array-like or sparse matrix, shape [n_samples, n_features]*) – Training data.
- **y** (*array-like, shape = [n_samples] or [n_samples, n_targets]*) – Target values.
- **sample_weight** (*array-like, shape = [n_samples]*) – Individual weights for each sample raises error if sample_weight is passed and base_estimator fit method does not support it.

Raises `ValueError` – If no valid consensus set could be found. This occurs if *is_data_valid* and *is_model_valid* return False for all *max_trials* randomly chosen sub-samples.

predict (*X*)

Predict using the estimated model.

This is a wrapper for *estimator_.predict(X)*.

Parameters **X** (*numpy array of shape [n_samples, n_features]*) –

Returns **y** – Returns predicted values.

Return type array, shape = [n_samples] or [n_samples, n_targets]

class `skll.learner.`**`RescaledRandomForestRegressor`**(*constrain=True*, *rescale=True*, ***kwargs*)

Bases: `sklearn.ensemble.forest.RandomForestRegressor`

`fit`(*X*, *y=None*)

Build a forest of trees from the training set (X, y).

Parameters

- **X** (*array-like or sparse matrix of shape = [n_samples, n_features]*) – The training input samples. Internally, its dtype will be converted to `dtype=np.float32`. If a sparse matrix is provided, it will be converted into a sparse `csc_matrix`.
- **y** (*array-like, shape = [n_samples] or [n_samples, n_outputs]*) – The target values (class labels in classification, real numbers in regression).
- **sample_weight** (*array-like, shape = [n_samples] or None*) – Sample weights. If None, then samples are equally weighted. Splits that would create child nodes with net zero or negative weight are ignored while searching for a split in each node. In the case of classification, splits are also ignored if they would result in any single class carrying a negative weight in either child node.

Returns **self**

Return type object

`predict`(*X*)

Predict regression target for X.

The predicted regression target of an input sample is computed as the mean predicted regression targets of the trees in the forest.

Parameters **X** (*array-like or sparse matrix of shape = [n_samples, n_features]*) – The input samples. Internally, its dtype will be converted to `dtype=np.float32`. If a sparse matrix is provided, it will be converted into a sparse `csr_matrix`.

Returns **y** – The predicted values.

Return type array of shape = [n_samples] or [n_samples, n_outputs]

class `skll.learner.`**`RescaledRidge`**(*constrain=True*, *rescale=True*, ***kwargs*)

Bases: `sklearn.linear_model.ridge.Ridge`

`fit`(*X*, *y=None*)

Fit Ridge regression model

Parameters

- **X** (*{array-like, sparse matrix}, shape = [n_samples, n_features]*) – Training data
- **y** (*array-like, shape = [n_samples] or [n_samples, n_targets]*) – Target values
- **sample_weight** (*float or numpy array of shape [n_samples]*) – Individual weights for each sample

Returns self

Return type returns an instance of self.

`predict`(*X*)

Predict using the linear model

Parameters X (*array_like or sparse matrix, shape (n_samples, n_features)*) – Samples.

Returns C – Returns predicted values.

Return type array, shape (n_samples,)

class `skll.learner.`**`RescaledSGDRegressor`**(*constrain=True*, *rescale=True*, ***kwargs*)

Bases: `sklearn.linear_model.stochastic_gradient.SGDRegressor`

`fit`(*X*, *y=None*)

Fit linear model with Stochastic Gradient Descent.

Parameters

- **X** (*{array-like, sparse matrix}, shape (n_samples, n_features)*) – Training data
- **y** (*numpy array, shape (n_samples,)*) – Target values
- **coef_init** (*array, shape (n_features,)*) – The initial coefficients to warm-start the optimization.
- **intercept_init** (*array, shape (1,)*) – The initial intercept to warm-start the optimization.
- **sample_weight** (*array-like, shape (n_samples,), optional*) – Weights applied to individual samples (1. for

unweighted).

Returns self

Return type returns an instance of self.

predict(*X*)

Predict using the linear model

Parameters X (*{array-like, sparse matrix}, shape (n_samples, n_features)*) –

Returns Predicted target values per element in X.

Return type array, shape (n_samples,)

class skll.learner.**RescaledSVR**(*constrain=True*, *rescale=True*, ***kwargs*)

Bases: sklearn.svm.classes.SVR

fit(*X*, *y=None*)

Fit the SVM model according to the given training data.

Parameters

- **X** (*{array-like, sparse matrix}, shape (n_samples, n_features)*) – Training vectors, where n_samples is the number of samples and n_features is the number of features. For kernel="precomputed", the expected shape of X is (n_samples, n_samples).
- **y** (*array-like, shape (n_samples,)*) – Target values (class labels in classification, real numbers in regression)
- **sample_weight** (*array-like, shape (n_samples,)*) – Per-sample weights. Rescale C per sample. Higher weights force the classifier to put more emphasis on these points.

Returns self

Return type object

Notes

If X and y are not C-ordered and contiguous arrays of np.float64 and X is not a scipy.sparse.csr_matrix, X and/or y may be copied.

If X is a dense array, then the other methods will not support sparse matrices as input.

predict(*X*)

Perform regression on samples in X.

For an one-class model, +1 (inlier) or -1 (outlier) is returned.

Parameters **X** (*{array-like, sparse matrix}, shape (n_samples, n_features)*) – For kernel="precomputed", the expected shape of X is (n_samples_test, n_samples_train).

Returns **y_pred**

Return type array, shape (n_samples,)

class `skll.learner.`**RescaledTheilSenRegressor**(*constrain=True*, *rescale=True*, ***kwargs*)

Bases: `sklearn.linear_model.theil_sen.TheilSenRegressor`

fit(*X*, *y=None*)

Fit linear model.

Parameters

- **X** (*numpy array of shape [n_samples, n_features]*) – Training data
- **y** (*numpy array of shape [n_samples]*) – Target values

Returns **self**

Return type returns an instance of self.

predict(*X*)

Predict using the linear model

Parameters **X** (*array_like or sparse matrix, shape (n_samples, n_features)*) – Samples.

Returns **C** – Returns predicted values.

Return type array, shape (n_samples,)

class `skll.learner.`**SelectByMinCount**(*min_count=1*)

Bases: `sklearn.feature_selection.univariate_selection.SelectKBest`

Select features occurring in more (and/or fewer than) than a specified number of examples in the training data (or a CV training fold).

Parameters **min_count** (*int, optional*) – The minimum feature count to select. Defaults to 1.

fit(*X*, *y=None*)

Fit the SelectByMinCount model.

Parameters

- **X** (*array-like, with shape (n_samples, n_features)*) – The training data to fit.

- **y** (*Ignored*) –

Returns

Return type self

`skll.learner.`**`rescaled`**(*cls*)

Decorator to create regressors that store a min and a max for the training data and make sure that predictions fall within that range. It also stores the means and SDs of the gold standard and the predictions on the training set to rescale the predictions (e.g., as in e-rater).

Parameters **cls** (*BaseEstimator*) – An estimator class to add rescaling to.

Returns **cls** – Modified version of estimator class with rescaled functions added.

Return type BaseEstimator

Raises `ValueError` – If classifier cannot be rescaled (i.e. is not a regressor).

1.6.6 `metrics` Module

This module contains a bunch of evaluation metrics that can be used to evaluate the performance of learners.

author Michael Heilman (mheilman@ets.org)

author Nitin Madnani (nmadnani@ets.org)

author Dan Blanchard (dblanchard@ets.org)

organization ETS

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

Calculate the F1 score of the least frequent label/class in `y_true` for `y_pred`.

Parameters

- **y_true** (*array-like of float*) – The true/actual/gold labels for the data.
- **y_pred** (*array-like of float*) – The predicted/observed labels for the data.

Returns **ret_score** – F1 score of the least frequent label.

Return type float

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

Calculates the kappa inter-rater agreement between two 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** (*array-like of float*) – The true/actual/gold labels for the data.
- **y_pred** (*array-like of float*) – The predicted/observed labels for the data.
- **weights** (*str or np.array, optional*) – Specifies the weight matrix for the calculation. Options are

```
-  None = unweighted-kappa
-  'quadratic' = quadratic-weighted kappa
-  'linear' = linear-weighted kappa
-  two-dimensional numpy array = a custom matrix of
```

  weights. Each weight corresponds to the w_{ij} values in the wikipedia description of how to calculate weighted Cohen's kappa. Defaults to None.
- **allow_off_by_one** (*bool, optional*) – 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. Defaults to False.

Returns **k** – The kappa score, or weighted 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.

`sklll.metrics.`**`kendall_tau`**(*y_true*, *y_pred*)

Calculate Kendall's tau between `y_true` and `y_pred`.

Parameters

- **y_true** (*array-like of float*) – The true/actual/gold labels for the data.

- **y_pred** (*array-like of float*) – The predicted/observed labels for the data.

Returns **ret_score** – Kendall's tau if well-defined, else 0.0

Return type float

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

Calculate Pearson product-moment correlation coefficient between `y_true` and `y_pred`.

Parameters

- **y_true** (*array-like of float*) – The true/actual/gold labels for the data.
- **y_pred** (*array-like of float*) – The predicted/observed labels for the data.

Returns **ret_score** – Pearson product-moment correlation coefficient if well-defined, else 0.0

Return type float

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

Calculate Spearman's rank correlation coefficient between `y_true` and `y_pred`.

Parameters

- **y_true** (*array-like of float*) – The true/actual/gold labels for the data.
- **y_pred** (*array-like of float*) – The predicted/observed labels for the data.

Returns **ret_score** – Spearman's rank correlation coefficient if well-defined, else 0.0

Return type float

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

Call the scoring function in `sklearn.metrics.SCORERS` with the given name. 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 from SCORERS.
- **y_true** (*array-like of float*) – The true/actual/gold labels for the data.

- **y_pred** (*array-like of float*) – The predicted/observed labels for the data.

Returns **ret_score** – The scored result from the given scorer.

Return type float

1.7 Internal Documentation

1.7.1 Release Process

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

1. Create a release branch on GitHub.
2. In the release branch:
   a. update the version numbers in `version.py`.
   b. update the conda recipe.
   c. update the documentation with any new features or details about changes.
   d. run `make linkcheck` on the documentation and fix any redirected/broken links.
   e. update the README.
3. Build the new conda package locally on your mac using the following command (*Note*: you may have to replace the contents of the `requirements()` function in `setup.py` with a `pass` statement to get `conda build` to work):

```
conda build -c defaults -c conda-forge --python=3.6 --numpy=1.14␣
↪skll
```

4. Convert the package for both linux and windows:

```
conda convert -p win-64 -p linux-64 <mac package tarball>
```

5. Upload each of the packages to anaconda.org using `anaconda upload <package tarball>`.
6. Upload source package to PyPI using `python setup.py sdist upload`.
7. Draft a release on GitHub.
8. Make a pull request with the release branch to be merged into `master` and request code review.
9. Once the build for the PR passes and the reviewers approve, merge the release branch into `master`.

10. Make sure that the RTFD build for `master` passes.
11. Tag the latest commit in `master` with the appropriate release tag and publish the release on GitHub.
12. Send an email around at ETS announcing the release and the changes.

CHAPTER 2

Indices and tables

- genindex
- modindex
- search

Python Module Index

Index