SciKit-Learn Laboratory Documentation

Release 1.3

Daniel Blanchard **Michael Heilman**
Nitin Madnani

Mar 04, 2017

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

Installation

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

```
pip install skll
```

or via `conda` (only for Python 3.4):

```
conda install -c desilinguist python=3.4 skll
```

It can also be downloaded directly from GitHub.

License

SKLL is distributed under the 3-clause BSD License.

Tutorial

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

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

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
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, the arguments to each of these 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:

```
Loading train/family.csv...           done
Loading train/misc.csv...           done
Loading train/socioeconomic.csv...           done
Loading train/vitals.csv...           done
Loading dev/family.csv...           done
Loading dev/misc.csv...           done
Loading dev/socioeconomic.csv...           done
Loading dev/vitals.csv...           done
2014-11-21 22:58:36,056 - skll.learner - WARNING - Training data will␣
 ↪be shuffled to randomize grid search folds.  Shuffling may yield␣
 ↪different results compared to scikit-learn.
Loading train/family.csv...           done
Loading train/misc.csv...           done
Loading train/socioeconomic.csv...           done
Loading train/vitals.csv...           done
Loading dev/family.csv...           done
Loading dev/misc.csv...           done
Loading dev/socioeconomic.csv...           done
Loading dev/vitals.csv...           done
2014-11-21 22:58:40,180 - skll.learner - WARNING - Training data will␣
 ↪be shuffled to randomize grid search folds.  Shuffling may yield␣
 ↪different results compared to scikit-learn.
Loading train/family.csv...           done
Loading train/misc.csv...           done
Loading train/socioeconomic.csv...           done
Loading train/vitals.csv...           done
Loading dev/family.csv...           done
Loading dev/misc.csv...           done
Loading dev/socioeconomic.csv...           done
Loading dev/vitals.csv...           done
```

```
2014-11-21 22:58:40,430 - skll.learner - WARNING - Training data will␣
↪be shuffled to randomize grid search folds.  Shuffling may yield␣
↪different results compared to scikit-learn.
Loading train/family.csv...           done
Loading train/misc.csv...           done
Loading train/socioeconomic.csv...           done
Loading train/vitals.csv...           done
Loading dev/family.csv...           done
Loading dev/misc.csv...           done
Loading dev/socioeconomic.csv...           done
Loading dev/vitals.csv...           done
2014-11-21 22:58:41,132 - skll.learner - WARNING - Training data will␣
↪be shuffled to randomize grid search folds.  Shuffling may yield␣
↪different results compared to scikit-learn.
```

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: 1.0.0
Training Set: train
Training Set Size: 712
Test Set: dev
Test Set Size: 179
Shuffle: False
Feature Set: ["family.csv", "misc.csv", "socioeconomic.csv", "vitals.
↪csv"]
```

```
Learner: RandomForestClassifier
Task: evaluate
Feature Scaling: none
Grid Search: True
Grid Search Folds: 3
Grid Objective Function: accuracy
Using Folds File: False
Scikit-learn Version: 0.15.2
Start Timestamp: 21 Nov 2014 22:58:35.940243
End Timestamp: 21 Nov 2014 22:58:40.072254
Total Time: 0:00:04.132011


Fold:
Model Parameters: {"max_depth": 10, "compute_importances": null, "min_
 ↪density": null, "bootstrap": true, "n_estimators": 500, "verbose": 0,
 ↪ "min_samples_split": 2, "max_features": "auto", "min_samples_leaf":␣
 ↪1, "criterion": "gini", "random_state": 123456789, "max_leaf_nodes":␣
 ↪null, "n_jobs": 1, "oob_score": 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
```

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*.

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.

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

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 nomimal 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.

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

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

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 Classifier. 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'`.
- **DecisionTreeClassifier**: Decision Tree Classifier
- **GradientBoostingClassifier**: Gradient Boosting Classifier
- **KNeighborsClassifier**: K-Nearest Neighbors Classifier
- **LinearSVC**: SVM using LibLinear
- **LogisticRegression**: Logistic regression using LibLinear
- **MultinomialNB**: Multinomial Naive Bayes
- **RandomForestClassifier**: Random Forest Classifier

- **SGDClassifier**: Stochastic Gradient Descent Classifier
- **SVC**: SVM using LibSVM

Regressors:

- **AdaBoostRegressor**: AdaBoost Regressor. 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`.
- **DecisionTreeRegressor**: Decision Tree Regressor
- **ElasticNet**: ElasticNet Regression
- **GradientBoostingRegressor**: Gradient Boosting Regressor
- **KNeighborsRegressor**: K-Nearest Neighbors Regressor
- **Lasso**: Lasso Regression
- **LinearRegression**: Linear Regression
- **LinearSVR**: Support Vector Regression using LibLinear
- **RandomForestRegressor**: Random Forest Regressor
- **Ridge**: Ridge Regression
- **SGDRegressor**: Stochastic Gradient Descent Regressor
- **SVR**: Support Vector Regression using LibSVM

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.

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`.

cv_folds_file *(Optional)*

Path to a csv file specifying folds for cross-validation. 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]

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`.

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

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 classifier in *learners* list. Any empty ones will be ignored (and the defaults will be used).

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

LogisticRegression

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

LinearSVC

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

SVC

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

DecisionTreeClassifier and DecisionTreeRegressor

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

RandomForestClassifier and RandomForestRegressor

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

GradientBoostingClassifier and GradientBoostingRegressor

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

SVR

```
{'cache_size': 1000, 'kernel': b'linear'}
```

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

Two possible options are available. The first one is `auto`, 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.

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.
- **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: Using `objective=x` instead of `objectives=['x']` is also acceptable, for backward-compatibility.

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]}]
```

DecisionTreeClassifier and DecisionTreeRegressor

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

ElasticNet, Lasso, and Ridge

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

GradientBoostingClassifier and GradientBoostingRegressor

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

KNeighborsClassifier and KNeighborsRegressor

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

LinearSVC

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

LogisticRegression

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

MultinomialNB

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

RandomForestClassifier and RandomForestRegressor

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

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]}]
```

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`.

results *(Optional)*

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

log *(Optional)*

Directory to store result 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*.

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`. 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.

`-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.[3]

-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`).

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.

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

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

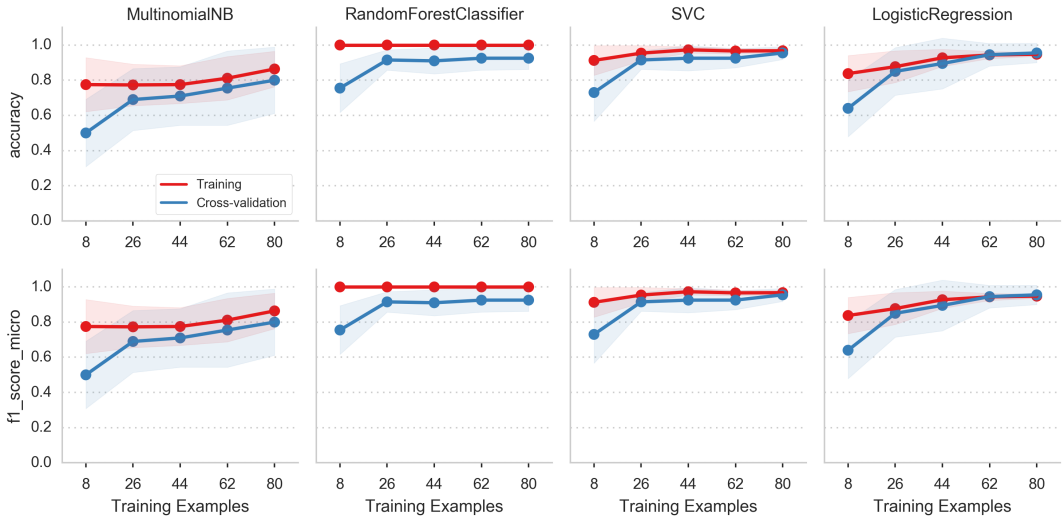
LEARNER The learner the current results/model/etc. was generated using.

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`.

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 pandas and seaborn are 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.



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.

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.

---

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.

---

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
A csv file, json file, or megam file (with or without the label column), with the appropriate suffix.

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)

-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.

---

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.

---

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.

---

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.

---

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.

API Documentation

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')
```

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)
```

`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.

> **Warning:** FeatureSets can only be equal if the order of the instances is identical because these are stored as lists/arrays.

This replaces `ExamplesTuple` from older versions.

Parameters

- **name** (*str*) – The name of this feature set.
- **ids** (*np.array*) – Example IDs for this set.
- **labels** (*np.array*) – labels for this set.
- **features** (*list of dict or array-like*) – The features for each instance represented as either a list of dictionaries or an array-like (if *vectorizer* is also specified).
- **vectorizer** (*DictVectorizer or FeatureHasher*) – Vectorizer which will be used to generate the feature matrix.

Note: 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 passed in parameters.

Parameters

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

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*) – Examples in the FeatureSet to keep. If *None*, no ID filtering takes place.

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

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

Helper function to create a FeatureSet object from a *pandas.DataFrame*. Will raise an Exception if pandas is not installed in your environment. *FeatureSet ids* will be the index on *df*.

Parameters

- **df** (*pandas.DataFrame*) – The pandas.DataFrame object you'd like to use as a feature set.
- **name** (*str*) – The name of this feature set.
- **labels_column** (*str or None*) – The name of the column containing the labels (data to predict).
- **vectorizer** (*DictVectorizer or FeatureHasher*) – Vectorizer which will be used to generate the feature matrix.

has_labels

Returns Whether or not this FeatureSet has any finite labels.

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

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

Parameters

- **fs** (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.

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*)

Bases: object

A little 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*) – Do not print "Loading..." status message to stderr.
- **ids_to_floats** (*bool*) – Convert IDs to float to save memory. Will raise error if we encounter an a non-numeric ID.
- **id_col** (*str*) – Name of the column which contains the instance IDs for ARFF/CSV/TSV files. If no column with that name exists, or *None* is specified, the IDs will be generated automatically.
- **label_col** (*str*) – 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.
- **class_map** (*dict from str to str*) – 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.
- **sparse** (*bool*) – Whether or not to store the features in a numpy CSR matrix when using a DictVectorizer to vectorize the features.
- **feature_hasher** (*bool*) – Whether or not a FeatureHasher should be used to vectorize the features.
- **num_features** (*int*) – 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.

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

Parameters

- **path** (*str or dict*) – The path to the file to load the examples from, or a list of example dictionaries.
- **quiet** (*bool*) – Do not print "Loading..." status message to stderr.

- **sparse** (*bool*) – Whether or not to store the features in a numpy CSR matrix.
- **id_col** (*str*) – Name of the column which contains the instance IDs for ARFF/CSV/TSV files. If no column with that name exists, or *None* is specified, the IDs will be generated automatically.
- **label_col** (*str*) – 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.
- **ids_to_floats** (*bool*) – Convert IDs to float to save memory. Will raise error if we encounter an a non-numeric ID.
- **class_map** (*dict from str to str*) – Mapping from original class labels to new ones. This is mainly used for collapsing multiple classes into a single class. Anything not in the mapping will be kept the same.

Returns New instance of the `Reader` sub-class that is appropriate for the given path, or `DictListReader` if given a list of dictionaries.

read()

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

Returns *FeatureSet* representing the file we read in.

class skll.**Writer**(*path*, *feature_set*, ***kwargs*)

Bases: `object`

Helper class for writing out FeatureSets to files.

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** (`FeatureSet`) – The FeatureSet to dump to a file.
- **quiet** (*bool*) – Do not print "Writing..." status message to stderr.
- **requires_binary** (*bool*) – Whether or not the Writer must open the file in binary mode for writing with Python 2.
- **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.

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

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`** (`FeatureSet`) – The FeatureSet to dump to a file.
- **`kwargs`** (*`dict`*) – The keyword arguments for `for_path` are the same as the initializer for the desired `Writer` subclass.

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

`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*)

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

Parameters

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

- **hosts** (*list of str*) – If running on the cluster, these are the machines we should use.
- **write_summary** (*bool*) – Write a tsv file with a summary of the results.
- **quiet** (*bool*) – Suppress printing of "Loading..." messages.
- **ablation** (*int or None*) – 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.
- **resume** (*bool*) – 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.

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

Return type list of str

From `learner` Module

class `skll.`**Learner**(*model_type*, *probability=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*)

Bases: `object`

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

Parameters

- **model_type** (*str*) – Type of estimator to create (e.g., LogisticRegression). See the skll package documentation for valid options.
- **probability** (*bool*) – Should learner return probabilities of all labels (instead of just label with highest probability)?
- **feature_scaling** (*str*) – how to scale the features, if at all. Options are: 'with_std': scale features using the standard deviation, 'with_mean': center features using the mean, 'both': do both scaling as well as centering, 'none': do neither scaling nor centering
- **model_kwargs** (*dict*) – A dictionary of keyword arguments to pass to the initializer for the specified model.
- **pos_label_str** (*str*) – The string for the positive label in the binary classification setting. Otherwise, an arbitrary label is picked.

- **min_feature_count** (*int*) – The minimum number of examples a feature must have a nonzero value in to be included.
- **sampler** (*str*) – The sampler to use for kernel approximation, if desired. Valid values are: `'AdditiveChi2Sampler'`, `'Nystroem'`, `'RBFSampler'`, and `'SkewedChi2Sampler'`.
- **sampler_kwargs** (*dict*) – A dictionary of keyword arguments to pass to the initializer for the specified sampler.
- **custom_learner_path** (*str*) – Path to module where a custom classifier is defined.

cross_validate(*examples*, *stratified=True*, *cv_folds=10*, *grid_search=False*, *grid_search_folds=3*, *grid_jobs=None*, *grid_objective=u'f1_score_micro'*, *prediction_prefix=None*, *param_grid=None*, *shuffle=False*, *save_cv_folds=False*)

Cross-validates a given model on the training examples.

Parameters

- **examples** (FeatureSet) – The data to cross-validate learner performance on.
- **stratified** (*bool*) – Should we stratify the folds to ensure an even distribution of labels for each fold?
- **cv_folds** (*int or dict*) – The number of folds to use for cross-validation, or a mapping from example IDs to folds.
- **grid_search** (*bool*) – Should we do grid search when training each fold? Note: This will make this take *much* longer.
- **grid_search_folds** (*int*) – The number of folds to use when doing the grid search (ignored if cv_folds is set to a dictionary mapping examples to folds).
- **grid_jobs** (*int*) – The number of jobs to run in parallel when doing the grid search. If unspecified or 0, the number of grid search folds will be used.
- **grid_objective** (*function*) – The objective function to use when doing the grid search.
- **param_grid** (*list of dicts mapping from strs to lists of parameter values*) – The parameter grid to search through for grid search. If unspecified, a default parameter grid will be used.
- **prediction_prefix** (*str*) – If saving the predictions, this is the prefix that will be used for the filename. It will be followed by ".predictions"

- **shuffle** (*bool*) – Shuffle examples before splitting into folds for CV.
- **save_cv_folds** (*bool*) – Whether to save the cv fold ids or not

Returns The confusion matrix, overall accuracy, per-label PRFs, and model parameters for each fold in one list, and another list with the grid search scores for each fold. Also return a dictionary containing the test-fold number for each id if save_cv_folds is True, otherwise None.

Return type (list of 4-tuples, list of float, dict)

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

Evaluates a given model on a given dev or test example set.

Parameters

- **examples** (FeatureSet) – The examples to evaluate the performance of the model on.
- **prediction_prefix** (*str*) – If saving the predictions, this is the prefix that will be used for the filename. It will be followed by ".predictions"
- **append** (*bool*) – Should we append the current predictions to the file if it exists?
- **grid_objective** (*function*) – The objective function that was used when doing the grid search.

Returns The confusion matrix, the overall accuracy, the per-label PRFs, the model parameters, and the grid search objective function score.

Return type 5-tuple

classmethod from_file(*learner_path*)

Returns New instance of Learner from the pickle at the specified path.

learning_curve(*examples*, *cv_folds=10*, *train_sizes=array([0.1, 0.325, 0.55, 0.775, 1.])*, *objective=u'f1_score_micro'*)

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.data.FeatureSet*) – The data to generate the learning curve on.
- **cv_folds** (*int*) – The number of folds to use for cross-validation with each training size

- **train_sizes** (*list of float or int*) – 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. (default: *np.linspace(0.1, 1.0, 5)*)
- **objective** (*string*) – The name of the objective function to use when computing the train and test scores for the learning curve. (default: 'f1_score_micro')

Returns The scores on the training sets, the scores on the test set, and the numbers of training examples used to generate the curve.

Return type (list of float, list of float, list of int)

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

Parameters **learner_path** (*str*) – The path to the 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 `LinearModel` (e.g., `Ridge`) regression and liblinear models.

Returns Labeled weights and (labeled if more than one) intercept value(s)

Return type tuple of (`weights`, `intercepts`), where `weights` is a dict and `intercepts` is a dictionary

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 data set

Parameters

- **examples** (`FeatureSet`) – The examples to predict the labels for.
- **prediction_prefix** (*str*) – If saving the predictions, this is the prefix that will be used for the filename. It will be followed by ".predictions"

- **append** (*bool*) – Should we append the current predictions to the file if it exists?
- **class_labels** (*bool*) – For classifier, should we convert class indices to their (str) labels?

Returns The predictions returned by the learner.

Return type array

probability
Should learner return probabilities of all labels (instead of just label with highest probability)?

save(*learner_path*)
Save the learner to a file.

Parameters **learner_path** (*str*) – The path to where you want to save the learner.

train(*examples*, *param_grid=None*, *grid_search_folds=3*, *grid_search=True*, *grid_objective=u'f1_score_micro'*, *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** (FeatureSet) – The examples to train the model on.
- **param_grid** (*list of dicts mapping from strs to lists of parameter values*) – The parameter grid to search through for grid search. If unspecified, a default parameter grid will be used.
- **grid_search_folds** (*int or dict*) – The number of folds to use when doing the grid search, or a mapping from example IDs to folds.
- **grid_search** (*bool*) – Should we do grid search?
- **grid_objective** (*function*) – The objective function to use when doing the grid search.
- **grid_jobs** (*int*) – The number of jobs to run in parallel when doing the grid search. If unspecified or 0, the number of grid search folds will be used.
- **shuffle** (*bool*) – Shuffle examples (e.g., for grid search CV.)
- **create_label_dict** (*bool*) – 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`.

Returns The best grid search objective function score, or 0 if we're not doing grid search.

Return type float

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 F1 score of the least frequent label

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 numpy array*) – 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.

- **allow_off_by_one** (*bool*) – 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.

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 Kendall's tau if well-defined, else 0

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 Spearman's rank correlation coefficient if well-defined, else 0

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 Pearson product-moment correlation coefficient if well-defined, else 0

data Package

data.featureset Module

Classes related to storing/merging feature sets.

author Dan Blanchard (dblanchard@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.

> **Warning:** FeatureSets can only be equal if the order of the instances is identical because these are stored as lists/arrays.

This replaces ExamplesTuple from older versions.

Parameters

- **name** (*str*) – The name of this feature set.
- **ids** (*np.array*) – Example IDs for this set.
- **labels** (*np.array*) – labels for this set.
- **features** (*list of dict or array-like*) – The features for each instance represented as either a list of dictionaries or an array-like (if *vectorizer* is also specified).
- **vectorizer** (*DictVectorizer or FeatureHasher*) – Vectorizer which will be used to generate the feature matrix.

Note: 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 passed in parameters.

Parameters

- **ids** (*list of str/float*) – Examples to keep in the FeatureSet. If *None*, no ID filtering takes place.
- **labels** (*list of str/float*) – labels that we want to retain examples for. If *None*, no label filtering takes place.

- **features** (*list of str*) – Features to keep in the FeatureSet. To help with filtering string-valued features that were converted to sequences of boolean features when read in, any features in the FeatureSet that contain a = will be split on the first occurrence and the prefix will be checked to see if it is in *features*. If *None*, no feature filtering takes place. Cannot be used if FeatureSet uses a FeatureHasher for vectorization.
- **inverse** (*bool*) – Instead of keeping features and/or examples in lists, remove them.

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

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

Helper function to create a FeatureSet object from a *pandas.DataFrame*. Will raise an Exception if pandas is not installed in your environment. *FeatureSet ids* will be the index on *df*.

Parameters

- **df** (*pandas.DataFrame*) – The pandas.DataFrame object you'd like to use as a feature set.
- **name** (*str*) – The name of this feature set.
- **labels_column** (*str or None*) – The name of the column containing the labels (data to predict).
- **vectorizer** (*DictVectorizer or FeatureHasher*) – Vectorizer which will be used to generate the feature matrix.

has_labels

Returns Whether or not this FeatureSet has any finite labels.

static split_by_ids (*fs*, *ids_for_split1*, *ids_for_split2=None*)
Split the given FeatureSet into two new FeatureSet instances based on the given IDs for the two splits.

Parameters

- **fs** (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.

data.readers 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 skll.data.readers.**ARFFReader** (*path_or_list*, ***kwargs*)
Bases: *skll.data.readers.DelimitedReader*

Reader for creating a *FeatureSet* out of an ARFF file.

If you would like to include example/instance IDs in your files, they must be specified as an "id" column.

Also, there must be a column with the name specified by *label_col* if the data is labelled, and this column must be the final one (as it is in Weka).

static split_with_quotes (*s*, *delimiter=u' '*, *quote_char=u"'"*, *escape_char=u'\\'*)
A replacement for string.split that won't split delimiters enclosed in quotes.

class skll.data.readers.**CSVReader** (*path_or_list*, ***kwargs*)
Bases: *skll.data.readers.DelimitedReader*

Reader for creating a *FeatureSet* out of a CSV file.

If you would like to include example/instance IDs in your files, they must be specified as an "id" column.

Also, there must be a column with the name specified by *label_col* if the data is labelled.

class skll.data.readers.**DelimitedReader**(*path_or_list*, ***kwargs*)
Bases: *skll.data.readers.Reader*

Reader for creating a *FeatureSet* out of a delimited (CSV/TSV) file.

If you would like to include example/instance IDs in your files, they must be specified as an `id` column.

Also, for ARFF, CSV, and TSV files, there must be a column with the name specified by *label_col* if the data is labelled. For ARFF files, this column must also be the final one (as it is in Weka).

Parameters **dialect** (*str*) – The dialect of to pass on to the underlying CSV reader. Default: `excel-tab`

class skll.data.readers.**DictListReader**(*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*)
Bases: *skll.data.readers.Reader*

This class is to facilitate programmatic use of *predict()* and other functions that take *FeatureSet* objects as input. It iterates over examples in the same way as other *Reader* clases, but uses a list of example dictionaries instead of a path to a file.

Parameters

- **path_or_list** (*Iterable of dict*) – List of example dictionaries.
- **quiet** (*bool*) – Do not print "Loading..." status message to stderr.
- **ids_to_floats** (*bool*) – Convert IDs to float to save memory. Will raise error if we encounter an a non-numeric ID.

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

Reader to create a *FeatureSet* out ouf 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*)

Bases: *skll.data.readers.Reader*

Reader to create a *FeatureSet* out ouf a MegaM -fvals file.

If you would like to include example/instance IDs in your files, they must be specified as a comment line directly preceding the line with feature values.

class skll.data.readers.**NDJReader**(*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*)

Bases: *skll.data.readers.Reader*

Reader to create a *FeatureSet* out of a .jsonlines/.ndj file

If you would like to include example/instance IDs in your files, they must be specified in the following ways as an "id" key in each JSON dictionary.

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*)

Bases: object

A little 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*) – Do not print "Loading..." status message to stderr.
- **ids_to_floats** (*bool*) – Convert IDs to float to save memory. Will raise error if we encounter an a non-numeric ID.
- **id_col** (*str*) – Name of the column which contains the instance IDs for ARFF/CSV/TSV files. If no column with that name exists, or *None* is specified, the IDs will be generated automatically.

- **label_col** (*str*) – 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.
- **class_map** (*dict from str to str*) – 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.
- **sparse** (*bool*) – Whether or not to store the features in a numpy CSR matrix when using a DictVectorizer to vectorize the features.
- **feature_hasher** (*bool*) – Whether or not a FeatureHasher should be used to vectorize the features.
- **num_features** (*int*) – 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.

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

Parameters

- **path** (*str or dict*) – The path to the file to load the examples from, or a list of example dictionaries.
- **quiet** (*bool*) – Do not print "Loading..." status message to stderr.
- **sparse** (*bool*) – Whether or not to store the features in a numpy CSR matrix.
- **id_col** (*str*) – Name of the column which contains the instance IDs for ARFF/CSV/TSV files. If no column with that name exists, or *None* is specified, the IDs will be generated automatically.
- **label_col** (*str*) – 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.
- **ids_to_floats** (*bool*) – Convert IDs to float to save memory. Will raise error if we encounter an a non-numeric ID.
- **class_map** (*dict from str to str*) – Mapping from original class labels to new ones. This is mainly used for collapsing multiple classes into a single class. Anything not in the mapping will be kept the same.

Returns New instance of the *Reader* sub-class that is appropriate for the given path, or *DictListReader* if given a list of dictionaries.

read()

Loads examples in the `.arff`, `.csv`, `.jsonlines`, `.libsvm`, `.megam`, `.ndj`, or

.tsv formats.

Returns *FeatureSet* representing the file we read in.

class skll.data.readers.**TSVReader**(*path_or_list*, ***kwargs*)
Bases: *skll.data.readers.DelimitedReader*

Reader for creating a *FeatureSet* out of a TSV file.

If you would like to include example/instance IDs in your files, they must be specified as an "id" column.

Also there must be a column with the name specified by *label_col* if the data is labelled.

skll.data.readers.**safe_float**(*text*, *replace_dict=None*)
Attempts to convert a string to an int, and then a float, but if neither is possible, just returns the original string value.

Parameters

- **text** (*str*) – The text to convert.
- **replace_dict** (*dict from str to str*) – 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.

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 skll.data.writers.**ARFFWriter**(*path*, *feature_set*, ***kwargs*)
Bases: *skll.data.writers.DelimitedFileWriter*

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** (FeatureSet) – The FeatureSet to dump to a file.

- **requires_binary** (*bool*) – Whether or not the Writer must open the file in binary mode for writing with Python 2.
- **quiet** (*bool*) – Do not print "Writing..." status message to stderr.
- **relation** (*str*) – The name of the relation in the ARFF file. Default: `'skll_relation'`
- **regression** (*bool*) – Is this an ARFF file to be used for regression? Default: `False`

class skll.data.writers.**CSVWriter**(*path*, *feature_set*, ***kwargs*)
Bases: *skll.data.writers.DelimitedFileWriter*

Writer for writing out FeatureSets 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** (FeatureSet) – The FeatureSet to dump to a file.
- **quiet** (*bool*) – Do not print "Writing..." status message to stderr.

class skll.data.writers.**DelimitedFileWriter**(*path*, *feature_set*, ***kwargs*)
Bases: *skll.data.writers.Writer*

Writer for writing out FeatureSets as TSV/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** (FeatureSet) – The FeatureSet to dump to a file.
- **quiet** (*bool*) – Do not print "Writing..." status message to stderr.
- **id_col** (*str*) – Name of the column to store the instance IDs in for ARFF, CSV, and TSV files.
- **label_col** (*str*) – Name of the column which contains the class labels for CSV/TSV files.
- **dialect** (*str*) – The dialect to use for the underlying `csv.DictWriter` Default: 'excel-tab'

class skll.data.writers.**LibSVMWriter**(*path*, *feature_set*, ***kwargs*)
Bases: *skll.data.writers.Writer*

Writer for writing out FeatureSets as LibSVM/SVMLight files.

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*, ***kwargs*)
Bases: *skll.data.writers.Writer*

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

class skll.data.writers.**TSVWriter**(*path*, *feature_set*, ***kwargs*)
Bases: *skll.data.writers.DelimitedFileWriter*

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** (`FeatureSet`) – The FeatureSet to dump to a file.
- **quiet** (*bool*) – Do not print "Writing..." status message to stderr.

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

Helper class for writing out FeatureSets to files.

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** (`FeatureSet`) – The FeatureSet to dump to a file.
- **quiet** (*bool*) – Do not print "Writing..." status message to stderr.
- **requires_binary** (*bool*) – Whether or not the Writer must open the file in binary mode for writing with Python 2.
- **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.

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

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`** (`FeatureSet`) – The FeatureSet to dump to a file.
- **`kwargs`** (*`dict`*) – The keyword arguments for `for_path` are the same as the initializer for the desired `Writer` subclass.

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

`write`()

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

`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 `skll.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: http://stackoverflow.com/questions/11561932/why-does-json-dumpslistnp-arange5-fail-while-json-dumpsnp-arange5-tolis

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*)

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

Parameters

- **config_path** (*str*) – Path to the configuration file we would like to use.
- **local** (*bool*) – Should this be run locally instead of on the cluster?
- **overwrite** (*bool*) – If the model files already exist, should we overwrite them instead of re-using them?
- **queue** (*str*) – The DRMAA queue to use if we're running on the cluster.
- **hosts** (*list of str*) – If running on the cluster, these are the machines we should use.
- **write_summary** (*bool*) – Write a tsv file with a summary of the results.
- **quiet** (*bool*) – Suppress printing of "Loading..." messages.
- **ablation** (*int or None*) – 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.
- **resume** (*bool*) – 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.

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

Return type list of str

`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 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.

class skll.learner.**Learner**(*model_type*, *probability=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*)
Bases: object

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

Parameters

- **model_type** (*str*) – Type of estimator to create (e.g., LogisticRegression). See the skll package documentation for valid options.
- **probability** (*bool*) – Should learner return probabilities of all labels (instead of just label with highest probability)?
- **feature_scaling** (*str*) – how to scale the features, if at all. Options are: 'with_std': scale features using the standard deviation, 'with_mean': center features using the mean, 'both': do both scaling as well as centering, 'none': do neither scaling nor centering
- **model_kwargs** (*dict*) – A dictionary of keyword arguments to pass to the initializer for the specified model.
- **pos_label_str** (*str*) – The string for the positive label in the binary classification setting. Otherwise, an arbitrary label is picked.
- **min_feature_count** (*int*) – The minimum number of examples a feature must have a nonzero value in to be included.
- **sampler** (*str*) – The sampler to use for kernel approximation, if desired. Valid values are: 'AdditiveChi2Sampler', 'Nystroem', 'RBFSampler', and 'SkewedChi2Sampler'.
- **sampler_kwargs** (*dict*) – A dictionary of keyword arguments to pass to the initializer for the specified sampler.

- **custom_learner_path** (*str*) – Path to module where a custom classifier is defined.

cross_validate(*examples*, *stratified=True*, *cv_folds=10*, *grid_search=False*, *grid_search_folds=3*, *grid_jobs=None*, *grid_objective=u'f1_score_micro'*, *prediction_prefix=None*, *param_grid=None*, *shuffle=False*, *save_cv_folds=False*)

Cross-validates a given model on the training examples.

Parameters

- **examples** (FeatureSet) – The data to cross-validate learner performance on.
- **stratified** (*bool*) – Should we stratify the folds to ensure an even distribution of labels for each fold?
- **cv_folds** (*int or dict*) – The number of folds to use for cross-validation, or a mapping from example IDs to folds.
- **grid_search** (*bool*) – Should we do grid search when training each fold? Note: This will make this take *much* longer.
- **grid_search_folds** (*int*) – The number of folds to use when doing the grid search (ignored if cv_folds is set to a dictionary mapping examples to folds).
- **grid_jobs** (*int*) – The number of jobs to run in parallel when doing the grid search. If unspecified or 0, the number of grid search folds will be used.
- **grid_objective** (*function*) – The objective function to use when doing the grid search.
- **param_grid** (*list of dicts mapping from strs to lists of parameter values*) – The parameter grid to search through for grid search. If unspecified, a default parameter grid will be used.
- **prediction_prefix** (*str*) – If saving the predictions, this is the prefix that will be used for the filename. It will be followed by ".predictions"
- **shuffle** (*bool*) – Shuffle examples before splitting into folds for CV.
- **save_cv_folds** (*bool*) – Whether to save the cv fold ids or not

Returns The confusion matrix, overall accuracy, per-label PRFs, and model parameters for each fold in one list, and another list with the grid search scores for each fold. Also return a dictionary containing the test-fold number for each id if save_cv_folds is True, otherwise None.

Return type (list of 4-tuples, list of float, dict)

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

Evaluates a given model on a given dev or test example set.

Parameters

- **examples** (FeatureSet) – The examples to evaluate the performance of the model on.
- **prediction_prefix** (*str*) – If saving the predictions, this is the prefix that will be used for the filename. It will be followed by ".predictions"
- **append** (*bool*) – Should we append the current predictions to the file if it exists?
- **grid_objective** (*function*) – The objective function that was used when doing the grid search.

Returns The confusion matrix, the overall accuracy, the per-label PRFs, the model parameters, and the grid search objective function score.

Return type 5-tuple

classmethod from_file(*learner_path*)

Returns New instance of Learner from the pickle at the specified path.

learning_curve(*examples*, *cv_folds=10*, *train_sizes=array([0.1, 0.325, 0.55, 0.775, 1.])*, *objective=u'f1_score_micro'*)

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.data.FeatureSet*) – The data to generate the learning curve on.
- **cv_folds** (*int*) – The number of folds to use for cross-validation with each training size
- **train_sizes** (*list of float or int*) – 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. (default: *np.linspace(0.1, 1.0, 5)*)

- **objective** (*string*) – The name of the objective function to use when computing the train and test scores for the learning curve. (default: 'f1_score_micro')

Returns The scores on the training sets, the scores on the test set, and the numbers of training examples used to generate the curve.

Return type (list of float, list of float, list of int)

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

Parameters **learner_path** (*str*) – The path to the 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 `LinearModel` (e.g., `Ridge`) regression and liblinear models.

Returns Labeled weights and (labeled if more than one) intercept value(s)

Return type tuple of (`weights`, `intercepts`), where `weights` is a dict and `intercepts` is a dictionary

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 data set

Parameters

- **examples** (`FeatureSet`) – The examples to predict the labels for.
- **prediction_prefix** (*str*) – If saving the predictions, this is the prefix that will be used for the filename. It will be followed by ".predictions"
- **append** (*bool*) – Should we append the current predictions to the file if it exists?
- **class_labels** (*bool*) – For classifier, should we convert class indices to their (str) labels?

Returns The predictions returned by the learner.

Return type array

probability
Should learner return probabilities of all labels (instead of just label with highest probability)?

save(*learner_path*)
Save the learner to a file.

Parameters **learner_path** (*str*) – The path to where you want to save the learner.

train(*examples*, *param_grid=None*, *grid_search_folds=3*, *grid_search=True*, *grid_objective=u'f1_score_micro'*, *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** (FeatureSet) – The examples to train the model on.
- **param_grid** (*list of dicts mapping from strs to lists of parameter values*) – The parameter grid to search through for grid search. If unspecified, a default parameter grid will be used.
- **grid_search_folds** (*int or dict*) – The number of folds to use when doing the grid search, or a mapping from example IDs to folds.
- **grid_search** (*bool*) – Should we do grid search?
- **grid_objective** (*function*) – The objective function to use when doing the grid search.
- **grid_jobs** (*int*) – The number of jobs to run in parallel when doing the grid search. If unspecified or 0, the number of grid search folds will be used.
- **shuffle** (*bool*) – Shuffle examples (e.g., for grid search CV.)
- **create_label_dict** (*bool*) – 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`.

Returns The best grid search objective function score, or 0 if we're not doing grid search.

Return type float

class skll.learner.**SelectByMinCount**(*min_count=1*)
Bases: sklearn.feature_selection.univariate_selection.SelectKBest

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

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*) – A regressor to add rescaling to.

Returns Modified version of class with rescaled functions added.

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 F1 score of the least frequent label

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 numpy array*) – 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.
- **allow_off_by_one** (*bool*) – 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.

skll.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 Kendall's tau if well-defined, else 0

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 Pearson product-moment correlation coefficient if well-defined, else 0

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 Spearman's rank correlation coefficient if well-defined, else 0

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.

CHAPTER 2

Indices and tables

- genindex
- modindex
- search

Python Module Index

S

Index

Symbols