
**SciKit-Learn Laboratory
Documentation**
Release 2.5.0

Educational Testing Service

Jul 01, 2020

Contents

1	Documentation	3
1.1	Installation	3
1.2	License	3
1.3	Tutorial	3
1.3.1	Workflow	4
1.3.2	Titanic Example	4
1.3.3	IRIS Example on Binder	10
1.4	Running Experiments	10
1.4.1	General Workflow	10
1.4.2	Feature files	11
1.4.3	Configuration file fields	13
1.4.4	Using run_experiment	34
1.4.5	Output files	36
1.5	Using Custom Metrics	38
1.5.1	Writing Custom Metric Functions	38
1.5.2	Using in Configuration Files	39
1.5.3	Using via the API	40
1.6	Utility Scripts	41
1.6.1	compute_eval_from_predictions	41
1.6.2	filter_features	42
1.6.3	generate_predictions	43
1.6.4	join_features	44
1.6.5	plot_learning_curves	45
1.6.6	print_model_weights	45
1.6.7	skll_convert	46
1.6.8	summarize_results	47
1.7	API Documentation	47
1.7.1	Quickstart	47
1.7.2	skll Package	48

1.7.3	config Package	60
1.7.4	data Package	61
1.7.5	experiments Package	73
1.7.6	learner Package	76
1.7.7	metrics Module	84
1.7.8	utils Package	86
1.8	Contributing	87
1.8.1	Guidelines	87
1.8.2	SKLL Code Overview	88
1.9	Internal Documentation	90
1.9.1	Release Process	90
2	Indices and tables	93
	Python Module Index	95
	Index	97



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*.

1.1 Installation

SKLL can be installed via `pip`:

```
pip install skll
```

or via `conda`:

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

It can also be downloaded directly from [GitHub](#).

1.2 License

SKLL is distributed under the 3-clause BSD License.

1.3 Tutorial

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

1.3.1 Workflow

In general, there are four steps to using SKLL:

1. Get some data in a *SKLL-compatible format*.
2. Create a small *configuration file* describing the machine learning experiment you would like to run.
3. Run that configuration file with *run_experiment*.
4. Examine the results of the experiment.

1.3.2 Titanic Example

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

Get your data into the correct format

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

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

Create a configuration file for the experiment

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

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

- Decision Tree
- Multinomial Naïve Bayes
- Random Forest
- Support Vector Machine


```

[General]
experiment_name = Titanic_Evaluate_Tuned
task = evaluate

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

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

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

```

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

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

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

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

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

Running your configuration file through `run_experiment`

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

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

That should produce output like:

```
2020-03-10 14:25:23,596 - Titanic_Evaluate_Tuned_family.csv+misc.
  ↳csv+socioeconomic.csv+vitals.csv_RandomForestClassifier - INFO -
  ↳Task: evaluate
2020-03-10 14:25:23,596 - Titanic_Evaluate_Tuned_family.csv+misc.
  ↳csv+socioeconomic.csv+vitals.csv_RandomForestClassifier - INFO -
  ↳Training on train, Test on dev, feature set ['family.csv', 'misc.csv
  ↳', 'socioeconomic.csv', 'vitals.csv'] ...
Loading /Users/nmadnani/work/skll/examples/titanic/train/family.csv...
  ↳ done
Loading /Users/nmadnani/work/skll/examples/titanic/train/misc.csv...
  ↳ done
Loading /Users/nmadnani/work/skll/examples/titanic/train/socioeconomic.
  ↳csv... done
Loading /Users/nmadnani/work/skll/examples/titanic/train/vitals.csv...
  ↳ done
Loading /Users/nmadnani/work/skll/examples/titanic/dev/family.csv...
  ↳ done
Loading /Users/nmadnani/work/skll/examples/titanic/dev/misc.csv...
  ↳ done
Loading /Users/nmadnani/work/skll/examples/titanic/dev/socioeconomic.
  ↳csv... done
Loading /Users/nmadnani/work/skll/examples/titanic/dev/vitals.csv...
  ↳ done
2020-03-10 14:25:23,662 - Titanic_Evaluate_Tuned_family.csv+misc.
  ↳csv+socioeconomic.csv+vitals.csv_RandomForestClassifier - INFO -
  ↳Featurizing and training new RandomForestClassifier model
```

(continues on next page)

(continued from previous page)

```

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

```

(continues on next page)

(continued from previous page)

```

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

```

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

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

Examine the results

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

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

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

```

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

Fold:
Model Parameters: {"bootstrap": true, "ccp_alpha": 0.0, "class_weight
↳": null, "criterion": "gini", "max_depth": 5, "max_features": "auto",
↳ "max_leaf_nodes": null, "max_samples": null, "min_impurity_decrease
↳": 0.0, "min_impurity_split": null, "min_samples_leaf": 1, "min_
↳samples_split": 2, "min_weight_fraction_leaf": 0.0, "n_estimators":
↳500, "n_jobs": null, "oob_score": false, "random_state": 123456789,
↳"verbose": 0, "warm_start": false}
Grid Objective Score (Train) = 0.797874315418175
+-----+-----+-----+-----+-----+-----+
|   |   0 |   1 | Precision | Recall | F-measure |
+====+====+====+====+====+====+
| 0 | [79] | 8 | 0.849 | 0.908 | 0.878 |
+-----+-----+-----+-----+-----+
| 1 | 14 | [42] | 0.840 | 0.750 | 0.792 |
+-----+-----+-----+-----+-----+
(row = reference; column = predicted)
Accuracy = 0.8461538461538461
Objective Function Score (Test) = 0.8461538461538461

Additional Evaluation Metrics (Test):
roc_auc = 0.9224137931034483

```

1.3.3 IRIS Example on Binder

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

1.4 Running Experiments

1.4.1 General Workflow

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

Get your data into the correct format

SKLL can work with several common data formats, all of which are described [here](#).

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

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

Create sparse feature files, if necessary

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

Set up training and testing directories/files

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

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

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

Create an experiment configuration file

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

Run configuration file through `run_experiment`

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

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

1.4.2 Feature files

SKLL supports the following feature file formats:

arff

The same file format used by [Weka](#) with the following added restrictions:

- Only simple numeric, string, and nominal values are supported.
- Nominal values are converted to strings.
- If the data has instance IDs, there should be an attribute with the name specified by `id_col` in the *Input* section of the configuration file you create for your experiment. This defaults to `i.d.` 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 configuration file you create for your experiment. This defaults to `y`. This must also be the final attribute listed (like in Weka).

csv/tsv

A simple comma or tab-delimited format. SKLL underlyingly uses [pandas](<https://pandas.pydata.org>) to read these files which is extremely fast but at the cost of some extra memory consumption.

When using this file format, the following restrictions apply:

- If the data is labelled, there must be a column with the name specified by `label_col` in the *Input* section of the configuration file you create for your experiment. This defaults to `y`.
- If the data has instance IDs, there should be a column with the name specified by `id_col` in the *Input* section of the configuration file you create for your experiment. This defaults to `i.d.` If there is no such column, IDs will be generated automatically.

- All other columns contain feature values, and every feature value must be specified (making this a poor choice for sparse data).

Warning:

1. SKLL will raise an error if there are blank values in **any** of the columns. You must either drop all rows with blank values in any column or replace the blanks with a value you specify. To drop or replace via the command line, use the `filter_features` script. You can also drop/replace via the SKLL Reader API, specifically `skll.data.readers.CSVReader` and `skll.data.readers.TSVReader`.
2. Dropping blanks will drop **all** rows with blanks in **any** of the columns. If you care only about **some** of the columns in the file and do not want to rows to be dropped due to blanks in the other columns, you should remove the columns you do not care about before dropping the blanks. For example, consider a hypothetical file `in.csv` that contains feature columns named A through G with the IDs stored in a column named ID and the labels stored in a column named CLASS. You only care about columns A, C, and F and want to drop all rows in the file that have blanks in any of these 3 columns but **do not** want to lose data due to there being blanks in any of the other columns. On the command line, you can run the following two commands:

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

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

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

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

jsonlines/ndj (Recommended)

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

following keys:

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

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

libsvm

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

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

The entire format would like this:

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

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

1.4.3 Configuration file fields

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

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

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

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

- If you want to **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 pipeline 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](#) under the `boston`, `iris`, and `titanic` sub-directories.

General

Both fields in the General section are required.

experiment_name

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

task

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

Input

The Input section must specify the machine learners to use via the *learners* field as well as the data and features to be used when training the model. This can be done by specifying either (a)

train_file in which case all of the features in the file will be used, or (b) *train_directory* along with *featuresets*.

learners

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

Classifiers:

- **AdaBoostClassifier:** [AdaBoost Classification](#). Note that the default base estimator is a `DecisionTreeClassifier`. A different base estimator can be used by specifying a `base_estimator` fixed parameter in the *fixed_parameters* list. The following additional base estimators are supported: `MultinomialNB`, `SGDClassifier`, and `SVC`. Note that the last two base require setting an additional `algorithm` fixed parameter with the value `'SAMME'`.
- **DummyClassifier:** [Simple rule-based Classification](#)
- **DecisionTreeClassifier:** [Decision Tree Classification](#)
- **GradientBoostingClassifier:** [Gradient Boosting Classification](#)
- **KNeighborsClassifier:** [K-Nearest Neighbors Classification](#)
- **LinearSVC:** [Support Vector Classification using LibLinear](#)
- **LogisticRegression:** [Logistic Regression Classification using LibLinear](#)
- **MLPClassifier:** [Multi-layer Perceptron Classification](#)
- **MultinomialNB:** [Multinomial Naive Bayes Classification](#)
- **RandomForestClassifier:** [Random Forest Classification](#)
- **RidgeClassifier:** [Classification using Ridge Regression](#)
- **SGDClassifier:** [Stochastic Gradient Descent Classification](#)
- **SVC:** [Support Vector Classification using LibSVM](#)

Regressors:

- **AdaBoostRegressor:** [AdaBoost Regression](#). Note that the default base estimator is a `DecisionTreeRegressor`. A different base estimator can be used by specifying a `base_estimator` fixed parameter in the *fixed_parameters* list. The following additional base estimators are supported: `SGDRegressor`, and `SVR`.
- **BayesianRidge:** [Bayesian Ridge Regression](#)
- **DecisionTreeRegressor:** [Decision Tree Regressor](#)

- **DummyRegressor**: Simple Rule-based Regression
- **ElasticNet**: ElasticNet Regression
- **GradientBoostingRegressor**: Gradient Boosting Regressor
- **HuberRegressor**: Huber Regression
- **KNeighborsRegressor**: K-Nearest Neighbors Regression
- **Lars**: Least Angle Regression
- **Lasso**: Lasso Regression
- **LinearRegression**: Linear Regression
- **LinearSVR**: Support Vector Regression using LibLinear
- **MLPRegressor**: Multi-layer Perceptron Regression
- **RandomForestRegressor**: Random Forest Regression
- **RANSACRegressor**: RANdom SAMple Consensus Regression. Note that the default base estimator is a `LinearRegression`. A different base regressor can be used by specifying a `base_estimator` fixed parameter in the *fixed_parameters* list.
- **Ridge**: Ridge Regression
- **SGDRegressor**: Stochastic Gradient Descent Regression
- **SVR**: Support Vector Regression using LibSVM
- **TheilSenRegressor**: Theil-Sen Regression

For all regressors, you can also prepend `Rescaled` to the beginning of the full name (e.g., `RescaledSVR`) to get a version of the regressor where predictions are rescaled and constrained to better match the training set.

featuresets

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

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

train_file

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

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

train_directory

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

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

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

class_map (Optional)

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

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

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

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

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

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

custom_learner_path (Optional)

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

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

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

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

custom_metric_path (Optional)

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

feature_hasher (Optional)

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

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

feature_scaling (*Optional*)

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

none Perform no feature scaling at all.

with_std Scale feature values by their standard deviation.

with_mean Center features by subtracting their mean.

both Perform both centering and scaling.

Defaults to none.

featureset_names (*Optional*)

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

folds_file (*Optional*)

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

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

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

The format of this file must be as follows: the first row must be a header. This header row is ignored, so it doesn't matter what the header row contains, but it must be there. If there is no header row, whatever row is in its place will be ignored. The first column should consist of training

set IDs and the second should be a string for the fold ID (e.g., 1 through 5, A through D, etc.). If specified, the CV and grid search will leave one fold ID out at a time.²

fixed_parameters (*Optional*)

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

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

AdaBoostClassifier and AdaBoostRegressor

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

DecisionTreeClassifier and DecisionTreeRegressor

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

DummyClassifier

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

ElasticNet

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

GradientBoostingClassifier and GradientBoostingRegressor

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

Lasso:

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

LinearSVC and LinearSVR

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

LogisticRegression

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

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

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

MLPClassifier and MLPRegressor:

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

RandomForestClassifier and RandomForestRegressor

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

RANSACRegressor

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

Ridge and RidgeClassifier

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

SVC and SVR

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

SGDClassifier

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

SGDRegressor

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

TheilSenRegressor

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

Note:

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

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

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

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

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

Additional examples and information can be seen [here](#).

hasher_features (*Optional*)

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

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

id_col (*Optional*)

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

ids_to_floats (*Optional*)

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

label_col (*Optional*)

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

learning_curve_cv_folds_list (*Optional*)

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

learning_curve_train_sizes (*Optional*)

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

num_cv_folds (*Optional*)

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

random_folds (*Optional*)

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

sampler (*Optional*)

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

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

sampler_parameters (Optional)

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

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

Nystroem

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

RBFSampler

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

SkewedChi2Sampler

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

shuffle (Optional)

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

suffix (Optional)

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

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

test_file (Optional)

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

test_directory (Optional)

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

Tuning

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

objectives

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

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

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

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

- **accuracy:** Overall accuracy
- **average_precision:** Area under PR curve . To use this metric, *probability* must be set to `True`. (*Binary classification only*).
- **balanced_accuracy:** A version of accuracy specifically designed for imbalanced binary and multi-class scenarios.
- **f1:** The default scikit-learn F_1 score (F_1 of the positive class for binary classification, or the weighted average F_1 for multiclass classification)
- **f1_score_macro:** Macro-averaged F_1 score
- **f1_score_micro:** Micro-averaged F_1 score

- **f1_score_weighted**: Weighted average F_1 score
- **f1_score_least_frequent**: F_1 score of the least frequent class. The least frequent class may vary from fold to fold for certain data distributions.
- **f05**: The default scikit-learn $F_{\beta=0.5}$ score ($F_{\beta=0.5}$ of the positive class for binary classification, or the weighted average $F_{\beta=0.5}$ for multiclass classification)
- **f05_score_macro**: Macro-averaged $F_{\beta=0.5}$ score
- **f05_score_micro**: Micro-averaged $F_{\beta=0.5}$ score
- **f05_score_weighted**: Weighted average $F_{\beta=0.5}$ score
- **jaccard**: The default Jaccard similarity coefficient from scikit-learn for binary classification.
- **jaccard_macro**: Macro-averaged Jaccard similarity coefficient
- **jaccard_micro**: Micro-averaged Jaccard similarity coefficient
- **jaccard_weighted**: Weighted average Jaccard similarity coefficient
- **kendall_tau**: Kendall's tau . For binary classification and with *probability* set to `True`, the probabilities for the positive class will be used to compute the correlation values. In all other cases, the labels are used. (*Integer labels only*).
- **linear_weighted_kappa**: Linear weighted kappa. (*Contiguous integer labels only*).
- **lwk_off_by_one**: Same as `linear_weighted_kappa`, but all ranking differences are discounted by one. (*Contiguous integer labels only*).
- **neg_log_loss**: The negative of the classification log loss . Since scikit-learn recommends using negated loss functions as scorer functions, SKLL does the same for the sake of consistency. To use this metric, *probability* must be set to `True`.
- **pearson**: Pearson correlation . For binary classification and with *probability* set to `True`, the probabilities for the positive class will be used to compute the correlation values. In all other cases, the labels are used. (*Integer labels only*).
- **precision**: Precision for binary classification
- **precision_macro**: Macro-averaged Precision
- **precision_micro**: Micro-averaged Precision
- **precision_weighted**: Weighted average Precision
- **quadratic_weighted_kappa**: Quadratic weighted kappa. (*Contiguous integer labels only*).
- **qwk_off_by_one**: Same as `quadratic_weighted_kappa`, but all ranking differences are discounted by one. (*Contiguous integer labels only*).

- **recall**: Recall for binary classification
- **recall_macro**: Macro-averaged Recall
- **recall_micro**: Micro-averaged Recall
- **recall_weighted**: Weighted average Recall
- **roc_auc**: Area under ROC curve .To use this metric, *probability* must be set to True. (*Binary classification only*).
- **spearman**: Spearman rank-correlation. For binary classification and with *probability* set to True, the probabilities for the positive class will be used to compute the correlation values. In all other cases, the labels are used. (*Integer labels only*).
- **unweighted_kappa**: Unweighted Cohen's kappa.
- **uwk_off_by_one**: Same as `unweighted_kappa`, but all ranking differences are discounted by one. In other words, a ranking of 1 and a ranking of 2 would be considered equal.

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

- **explained_variance**: A score indicating how much of the variance in the given data can be by the model.
- **kendall_tau**: Kendall's tau
- **linear_weighted_kappa**: Linear weighted kappa (any floating point values are rounded to ints)
- **lwk_off_by_one**: Same as `linear_weighted_kappa`, but all ranking differences are discounted by one.
- **max_error**: The maximum residual error.
- **neg_mean_absolute_error**: The negative of the mean absolute error regression loss. Since scikit-learn recommends using negated loss functions as scorer functions, SKLL does the same for the sake of consistency.
- **neg_mean_squared_error**: The negative of the mean squared error regression loss. Since scikit-learn recommends using negated loss functions as scorer functions, SKLL does the same for the sake of consistency.
- **pearson**: Pearson correlation
- **quadratic_weighted_kappa**: Quadratic weighted kappa (any floating point values are rounded to ints)
- **qwk_off_by_one**: Same as `quadratic_weighted_kappa`, but all ranking differences are discounted by one.
- **r2**: R2

- **spearman**: Spearman rank-correlation
- **unweighted_kappa**: Unweighted Cohen's kappa (any floating point values are rounded to ints)
- **uwk_off_by_one**: Same as `unweighted_kappa`, but all ranking differences are discounted by one. In other words, a ranking of 1 and a ranking of 2 would be considered equal.

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

grid_search (*Optional*)

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

Note:

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

grid_search_folds (*Optional*)

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

param_grids (Optional)

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

The default parameter grids for each learner are:

AdaBoostClassifier and AdaBoostRegressor

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

BayesianRidge

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

DecisionTreeClassifier and DecisionTreeRegressor

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

ElasticNet

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

GradientBoostingClassifier and GradientBoostingRegressor

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

HuberRegressor

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

KNeighborsClassifier and KNeighborsRegressor

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

Lasso

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

LinearSVC

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

LogisticRegression

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

MLPClassifier and MLPRegressor:

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

MultinomialNB

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

RandomForestClassifier and RandomForestRegressor

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

Ridge and RidgeClassifier

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

SGDClassifier and SGDRegressor

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

SVC

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

SVR

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

Note: Note that learners not listed here do not have any default parameter grids in SKLL either because there are no hyper-parameters to tune or decisions about which parameters to tune (and how) depend on the data being used for the experiment and are best left up to the user.

pos_label_str (Optional)

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

use_folds_file_for_grid_search (Optional)

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

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

Output

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

log (Optional)

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

models (Optional)

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

metrics (Optional)

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

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

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

pipeline (Optional)

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

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

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

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

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

```
from sklearn.preprocessing import LabelEncoder

from skll import Learner
from skll.data import Reader

# train a classifier and a regressor using the SKLL API
fs1 = Reader.for_path('examples/iris/train/example_iris_features.
→jsonlines').read()
learner1 = Learner('LogisticRegression', pipeline=True)
_ = learner1.train(fs1, grid_search=True, grid_objective='f1_score_
→macro')

fs2 = Reader.for_path('examples/boston/train/example_boston_features.
→jsonlines').read()
```

(continues on next page)

(continued from previous page)

```

learner2 = Learner('RescaledSVR', feature_scaling='both',
    →pipeline=True)
_ = learner2.train(fs2, grid_search=True, grid_objective='pearson')

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

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

# then, the regressor
D2 = {"f0": 0.09178, "f1": 0.0, "f2": 4.05, "f3": 0.0, "f4": 0.51, "f5"
    →": 6.416, "f6": 84.1, "f7": 2.6463, "f8": 5.0, "f9": 296.0, "f10":
    →16.6, "f11": 395.5, "f12": 9.04}
pipeline2 = learner2.pipeline
pipeline2.predict(D2)

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

```

Note:

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

predictions (*Optional*)

Directory in which to store *prediction files*. Can be omitted to not store predictions. Must *not* be specified for the *learning_curve* and *train* tasks.

probability (*Optional*)

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

results (*Optional*)

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

save_cv_folds (*Optional*)

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

save_cv_models (*Optional*)

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

1.4.4 Using `run_experiment`

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

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

Runs an ablation study where repeated experiments are conducted with the specified number of feature files in each featureset in the configuration file held out. For example, if you have three feature files (A, B, and C) in your featureset and you specify `--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]]`.

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

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

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

-A, --ablation_all

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

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

-k, --keep-models

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

-r, --resume

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

-v, --verbose

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

--version

Show program's version number and exit.

GridMap options

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

-l, --local

Run jobs locally instead of using the cluster.⁴

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

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

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

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

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

⁴ This will happen automatically if `GridMap` cannot be imported.

1.4.5 Output files

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

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

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

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

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

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

However, if the *objectives* field in the configuration file contains only a single value, the job can be disambiguated using only the featuresets and the learners since the objective is fixed. Therefore, the output files will have the prefix `<EXPERIMENT>_<FEATURESET>_<LEARNER>`.

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

Log files

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

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

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

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

Model files

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

Results files

SKLL generates two types of result files:

1. Files ending in `.results` which contain a human-readable summary of the job, complete with confusion matrix, objective function score on the test set, and values of any additional metrics specified via the `metrics` configuration file option.
2. Files ending in `.results.json`, which contain all of the same information as the `.results` files, but in a format more well-suited to automated processing. In some cases, `.results.json` files may contain *more* information than their `.results` file counterparts. For example, when doing `grid search` for tuning model hyperparameters, these files contain an additional attribute `grid_search_cv_results` containing detailed results from the grid search process.

Prediction files

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

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

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

Summary file

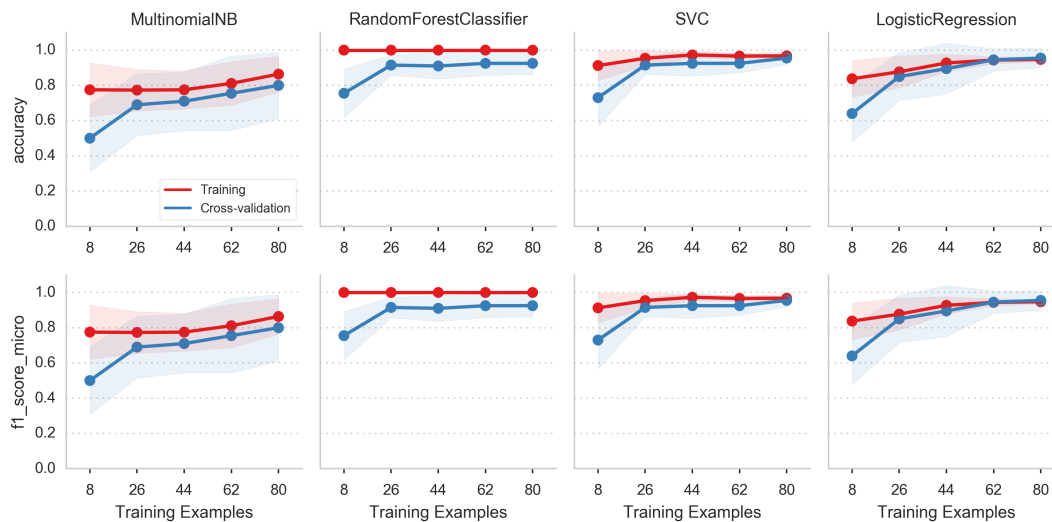
For every experiment you run, there will also be an experiment summary file generated that is a tab-delimited file summarizing the results for each job in the experiment. It is named `<EXPERIMENT>_summary.tsv`. For `learning_curve` experiments, this summary file will contain training set sizes and the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

Folds file

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

Learning curve plots

When running a *learning_curve* experiment, actual learning curves are also generated as PNG files - 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 feature-set with objective functions on rows and learners on columns. Here's an example of such a plot.



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

1.5 Using Custom Metrics

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

1.5.1 Writing Custom Metric Functions

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

1. `greater_is_better`: a boolean keyword argument that indicates whether a higher value of the metric indicates better performance (`True`) or vice versa (`False`). The default value is `True`.
2. `needs_proba`: a boolean keyword argument that indicates whether the metric function requires probability estimates. The default value is `False`.
3. `needs_threshold`: a boolean keyword argument that indicates whether the metric function takes a continuous decision certainty. The default value is `False`.

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

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

Listing 1: `custom.py`

```
from sklearn.metrics import fbeta_score

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

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

1.5.2 Using in Configuration Files

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

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

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

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

[Input]
```

(continues on next page)

(continued from previous page)

```

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

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

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

```

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

1.5.3 Using via the API

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

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

```

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

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

# let's assume the training data lives in a file called "train.csv"
# we load that into a SKLL FeatureSet

```

(continues on next page)

(continued from previous page)

```
fs = CSVReader.for_path("train.csv").read()

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

...
```

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

Note:

1. When using the API, if you have multiple metric functions defined in a Python source file, you must register each one individually using `register_custom_metric()`.
2. When using the API, if you try to re-register the same metric in the same Python session, it will raise a `NameError`. Therefore, if you edit your custom metric, you must start a new Python session to be able to see the changes.

Warning:

1. When using a configuration file or the API, if the names of any of your custom metric functions conflict with names of *metrics* that already exist in either SKLL or scikit-learn, it will raise a `NameError`. You should rename the metric function in that case.
2. Unlike for the built-in metrics, SKLL does not check whether your custom metric function is appropriate for classification or regression. You must make that decision for yourself.

1.6 Utility Scripts

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

1.6.1 `compute_eval_from_predictions`

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

Positional Arguments

examples_file

SKLL input file with labeled examples

predictions_file

file with predictions from SKLL

metric_names

metrics to compute

Optional Arguments

--version

Show program's version number and exit.

1.6.2 filter_features

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

Positional Arguments

infile

Input feature file (ends in `.arff`, `.csv`, `.jsonlines`, `.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.

--id_col <id_col>

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

- L** <label <label ...>>, **--label** <label <label ...>>
A label in the feature file you would like to keep. If unspecified, no instances are removed based on their labels.
 - l** <label_col>, **--label_col** <label_col>
Name of the column which contains the class labels in ARFF, CSV, or TSV files. For ARFF files, this must be the final column to count as the label. (default: *y*)
 - db**, **--drop-blanks**
Drop all lines/rows that have any blank values. (default: *False*)
 - rb** <replacement>, **--replace-blanks-with** <replacement>
Specifies a new value with which to replace blank values in all columns in the file. To replace blanks differently in each column, use the SKLL Reader API directly. (default: *None*)
 - q**, **--quiet**
Suppress printing of "Loading..." messages.
 - version**
Show program's version number and exit.
-

1.6.3 generate_predictions

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

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

Positional Arguments

model_file

Model file to load and use for generating predictions.

input_file(s)

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

Optional Arguments

- i** <id_col>, **--id_col** <id_col>
Name of the column which contains the instance IDs in ARFF, CSV, or TSV files. (default: id)
 - l** <label_col>, **--label_col** <label_col>
Name of the column which contains the labels in ARFF, CSV, or TSV files. For ARFF files, this must be the final column to count as the label. (default: y)
 - o** <path>, **--output_file** <path>
Path to output TSV file. If not specified, predictions will be printed to stdout. For probabilistic binary classification, the probability of the positive class will always be in the last column.
 - p**, **--predict_labels**
If the model does probabilistic classification, output the class label with the highest probability instead of the class probabilities.
 - q**, **--quiet**
Suppress printing of "Loading..." messages.
 - t** <threshold>, **--threshold** <threshold>
If the model does binary probabilistic classification, return the positive class label only if it meets/exceeds the given threshold and the other class label otherwise.
 - version**
Show program's version number and exit.
-

1.6.4 join_features

Combine multiple feature files into one larger file.

Positional Arguments

- infile** ...
Input feature files (ends in .arff, .csv, .jsonlines, .ndj, or .tsv)
- outfile**
Output feature file (must have same extension as input file)

Optional Arguments

- l** <label_col>, **--label_col** <label_col>
Name of the column which contains the labels in ARFF, CSV, or TSV files. For ARFF files, this must be the final column to count as the label. (default: y)
 - q**, **--quiet**
Suppress printing of "Loading..." messages.
 - version**
Show program's version number and exit.
-

1.6.5 plot_learning_curves

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

Positional Arguments

- tsv_file**
Input learning Curve TSV output file.
 - output_dir**
Output directory to store the learning curve plots.
-

1.6.6 print_model_weights

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

Positional Arguments

- model_file**
Model file to load.

Optional Arguments

- k** <k>
Number of top features to print (0 for all) (default: 50)

- sign** {positive,negative,all}
Show only positive, only negative, or all weights (default: all)
 - sort_by_labels**
Order the features by classes (default: False). Mutually exclusive with the --k option.
 - version**
Show program's version number and exit.
-

1.6.7 skill_convert

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

Positional Arguments

infile

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

outfile

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

Optional Arguments

- l** <label_col>, **--label_col** <label_col>
Name of the column which contains the labels in ARFF, CSV, or TSV files. For ARFF files, this must be the final column to count as the label. (default: y)
- q**, **--quiet**
Suppress printing of "Loading..." messages.
- arff_regression**
Create ARFF files for regression, not classification.
- arff_relation** ARFF_RELATION
Relation name to use for ARFF file. (default: skill_relation)
- no_labels**
Used to indicate that the input data has no labels.
- reuse_libsvm_map** REUSE_LIBSVM_MAP
If you want to output multiple files that use the same mapping from labels and features to numbers when writing libsvm files, you can specify an existing .libsvm file to reuse the mapping from.
- version**
Show program's version number and exit.

1.6.8 summarize_results

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

Positional Arguments

summary_file

TSV file to store summary of results.

json_file

JSON results file generated by *run_experiment*.

Optional Arguments

-a, --ablation

The results files are from an ablation run.

--version

Show program's version number and exit.

1.7 API Documentation

1.7.1 Quickstart

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

Load a `FeatureSet` from a file:

```
from skll.data import Reader

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

Or, work with an existing pandas `DataFrame`:

```
from skll 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)
```

1.7.2 skll Package

We have made the most useful parts of our API available in the top-level `skll` namespace even though some of them actually live in subpackages. They are documented in both places for convenience.

From data Package

```
class skll.FeatureSet(name, ids, labels=None, features=None, vector-
    izer=None)
```

Bases: `object`

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

Parameters

- **name** (*str*) – The name of this feature set.
- **ids** (*np.array*) – Example IDs for this set.

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

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

Notes

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

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

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

Parameters

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

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

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

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

Parameters

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

Yields

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

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

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

Helper function to create a `FeatureSet` instance from a `pandas.DataFrame`. Will raise an `Exception` if `pandas` is not installed in your environment. The `ids` in the `FeatureSet` will be the index from the given frame.

Parameters

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

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

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

Return type *skll.FeatureSet*

has_labels

Check if `FeatureSet` has finite labels.

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

Return type `bool`

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

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

Parameters

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

Returns

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

From experiments Package

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

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

Parameters

- **config_file** (*str*) – Path to the configuration file we would like to use.

- **local** (*bool, optional*) – Should this be run locally instead of on the cluster? Defaults to `False`.
- **overwrite** (*bool, optional*) – If the model files already exist, should we overwrite them instead of re-using them? Defaults to `True`.
- **queue** (*str, optional*) – The DRMAA queue to use if we’re running on the cluster. Defaults to `'all.q'`.
- **hosts** (*list of str, optional*) – If running on the cluster, these are the machines we should use. Defaults to `None`.
- **write_summary** (*bool, optional*) – Write a TSV file with a summary of the results. Defaults to `True`.
- **quiet** (*bool, optional*) – Suppress printing of “Loading…” messages. Defaults to `False`.
- **ablation** (*int, optional*) – Number of features to remove when doing an ablation experiment. If positive, we will perform repeated ablation runs for all combinations of features removing the specified number at a time. If `None`, we will use all combinations of all lengths. If 0, the default, no ablation is performed. If negative, a `ValueError` is raised. Defaults to 0.
- **resume** (*bool, optional*) – If result files already exist for an experiment, do not overwrite them. This is very useful when doing a large ablation experiment and part of it crashes. Defaults to `False`.
- **log_level** (*str, optional*) – The level for logging messages. Defaults to `logging.INFO`.

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

Return type list of str

Raises

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

From learner Package

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

Bases: `object`

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

Parameters

- **model_type** (*str*) – Name of estimator to create (e.g., 'LogisticRegression'). See the skill package documentation for valid options.
- **probability** (*bool, optional*) – Should learner return probabilities of all labels (instead of just label with highest probability)? Defaults to `False`.
- **pipeline** (*bool, optional*) – Should learner contain a pipeline attribute that contains a scikit-learn Pipeline object composed of all steps including the vectorizer, the feature selector, the sampler, the feature scaler, and the actual estimator. Note that this will increase the size of the learner object in memory and also when it is saved to disk. Defaults to `False`.
- **feature_scaling** (*str, optional*) – How to scale the features, if at all. Options are - 'with_std': scale features using the standard deviation - 'with_mean': center features using the mean - 'both': do both scaling as well as centering - 'none': do neither scaling nor centering Defaults to 'none'.
- **model_kwargs** (*dict, optional*) – A dictionary of keyword arguments to pass to the initializer for the specified model. Defaults to `None`.
- **pos_label_str** (*str, optional*) – A string denoting the label of the class to be treated as the positive class in a binary classification setting. If `None`, the class represented by the label that appears second when sorted is chosen as the positive class. For example, if the two labels in data are "A" and "B" and `pos_label_str` is not specified, "B" will be chosen as the positive class. Defaults to `None`.
- **min_feature_count** (*int, optional*) – The minimum number of examples a feature must have a nonzero value in to be included. Defaults to 1.
- **sampler** (*str, optional*) – The sampler to use for kernel approximation, if desired. Valid values are - 'AdditiveChi2Sampler' - 'Nystroem' - 'RBFSampler' - 'SkewedChi2Sampler' Defaults to `None`.
- **sampler_kwargs** (*dict, optional*) – A dictionary of keyword arguments to pass to the initializer for the specified sampler. Defaults to `None`.
- **custom_learner_path** (*str, optional*) – Path to module where a custom classifier is defined. Defaults to `None`.

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

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

Cross-validates a given model on the training examples.

Parameters

- **examples** (`skll.FeatureSet`) – The `FeatureSet` instance to cross-validate learner performance on.
- **stratified** (*bool, optional*) – Should we stratify the folds to ensure an even distribution of labels for each fold? Defaults to `True`.
- **cv_folds** (*int or dict, optional*) – The number of folds to use for cross-validation, or a mapping from example IDs to folds. Defaults to 10.
- **grid_search** (*bool, optional*) – Should we do grid search when training each fold? Note: This will make this take *much* longer. Defaults to `False`.
- **grid_search_folds** (*int or dict, optional*) – The number of folds to use when doing the grid search, or a mapping from example IDs to folds. Defaults to 3.
- **grid_jobs** (*int, optional*) – The number of jobs to run in parallel when doing the grid search. If `None` or 0, the number of grid search folds will be used. Defaults to `None`.
- **grid_objective** (*str, optional*) – The name of the objective function to use when doing the grid search. Must be specified if `grid_search` is `True`. Defaults to `None`.
- **output_metrics** (*list of str, optional*) – List of additional metric names to compute in addition to the metric used for grid search. Empty by default. Defaults to an empty list.
- **prediction_prefix** (*str, optional*) – If saving the predictions, this is the prefix that will be used for the filename. It will be followed by `"_predictions.tsv"`. Defaults to `None`.
- **param_grid** (*dict, optional*) – The parameter grid to search. Defaults to `None`.

- **shuffle** (*bool, optional*) – Shuffle examples before splitting into folds for CV. Defaults to `False`.
- **save_cv_folds** (*bool, optional*) – Whether to save the cv fold ids or not? Defaults to `False`.
- **save_cv_models** (*bool, optional*) – Whether to save the cv models or not? Defaults to `False`.
- **use_custom_folds_for_grid_search** (*bool, optional*) – If `cv_folds` is a custom dictionary, but `grid_search_folds` is not, perhaps due to user oversight, should the same custom dictionary automatically be used for the inner grid-search cross-validation? Defaults to `True`.

Returns

- **results** (*list of 6-tuples*) – The confusion matrix, overall accuracy, per-label PRFs, model parameters, objective function score, and evaluation metrics (if any) for each fold.
- **grid_search_scores** (*list of floats*) – The grid search scores for each fold.
- **grid_search_cv_results_dicts** (*list of dicts*) – A list of dictionaries of grid search CV results, one per fold, with keys such as “params”, “mean_test_score”, etc, that are mapped to lists of values associated with each hyperparameter set combination.
- **skll_fold_ids** (*dict*) – A dictionary containing the test-fold number for each id if `save_cv_folds` is `True`, otherwise `None`.
- **models** (*list of skll.learner.Learner*) – A list of `skll.learner.Learners`, one for each fold if `save_cv_models` is `True`, otherwise `None`.

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

evaluate (*examples*, *prediction_prefix=None*, *append=False*, *grid_objective=None*, *output_metrics=[]*)
Evaluates a given model on a given dev or test `FeatureSet`.

Parameters

- **examples** (`skll.FeatureSet`) – The `FeatureSet` instance to evaluate the performance of the model on.
- **prediction_prefix** (*str, optional*) – If not `None`, predictions will also be written out to a file with the name `<prediction_prefix>_predictions.tsv`. Note that the prefix can also contain a path. Defaults to `None`.

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

Returns res – The confusion matrix, the overall accuracy, the per-label PRFs, the model parameters, the grid search objective function score, and the additional evaluation metrics, if any. For regressors, the first two elements are `None`.

Return type 6-tuple

classmethod from_file (*learner_path, logger=None*)

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

Parameters

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

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

Return type *skll.Learner*

Raises

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

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

Generates learning curves for a given model on the training examples via cross-validation. Adapted from the scikit-learn code for learning curve generation (cf. “`sklearn.model_selection.learning_curve`”).

Parameters

- **examples** (*skll.FeatureSet*) – The `FeatureSet` instance to generate the learning curve on.
- **cv_folds** (*int or dict, optional*) – The number of folds to use for cross-validation, or a mapping from example IDs to folds. Defaults to 10.

- **metric** (*str*) – The name of the metric function to use when computing the train and test scores for the learning curve.
- **train_sizes** (*list of float or int, optional*) – Relative or absolute numbers of training examples that will be used to generate the learning curve. If the type is float, it is regarded as a fraction of the maximum size of the training set (that is determined by the selected validation method), i.e. it has to be within (0, 1]. Otherwise it is interpreted as absolute sizes of the training sets. Note that for classification the number of samples usually have to be big enough to contain at least one sample from each class. Defaults to `np.linspace(0.1, 1.0, 5)`.

Returns

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

load (*learner_path*)

Replace the current learner instance with a saved learner.

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

model

The underlying scikit-learn model

model_kwargs

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

model_params

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

Returns

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

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

model_type

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

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

Uses a given model to return, and optionally, write out predictions on a given `FeatureSet` to a file.

For regressors, the returned and written-out predictions are identical. However, for classifiers: - if `class_labels` is `True`, class labels are returned

as well as written out.

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

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

Parameters

- **examples** (`skll.FeatureSet`) – The `FeatureSet` instance to predict labels for.
- **prediction_prefix** (`str`, *optional*) – If not `None`, predictions will also be written out to a file with the name `<prediction_prefix>_predictions.tsv`. For classifiers, the predictions written out are class labels unless the learner is probabilistic AND `class_labels` is set to `False`. Note that this prefix can also contain a path. Defaults to `None`.
- **append** (`bool`, *optional*) – Should we append the current predictions to the file if it exists? Defaults to `False`.
- **class_labels** (`bool`, *optional*) – If `False`, return either the class probabilities (probabilistic classifiers) or the class indices (non-probabilistic ones). If `True`, return the class labels no matter what. Ignored for regressors. Defaults to `True`.

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

Return type array-like

Raises

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

probability

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

save (*learner_path*)

Save the `Learner` instance to a file.

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

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

Train the model underlying the learner and return the grid search score and a dictionary of grid search results.

Parameters

- **examples** (`skll.FeatureSet`) – The `FeatureSet` instance to use for training.
- **param_grid** (*dict*, *optional*) – The parameter grid to search through for grid search. If `None`, a default parameter grid will be used. Defaults to `None`.
- **grid_search_folds** (*int or dict*, *optional*) – The number of folds to use when doing the grid search, or a mapping from example IDs to folds. Defaults to 3.
- **grid_search** (*bool*, *optional*) – Should we do grid search? Defaults to `True`.
- **grid_objective** (*str*, *optional*) – The name of the objective function to use when doing the grid search. Must be specified if `grid_search` is `True`. Defaults to `None`.
- **grid_jobs** (*int*, *optional*) – The number of jobs to run in parallel when doing the grid search. If `None` or 0, the number of grid search folds will be used. Defaults to `None`.
- **shuffle** (*bool*, *optional*) – Shuffle examples (e.g., for grid search CV.) Defaults to `False`.

Returns tuple – 1) The best grid search objective function score, or 0 if we’re not doing grid search, and 2) a dictionary of grid search CV results with keys such as “`params`”, “`mean_test_score`”, etc, that are mapped to lists of values associated with each hyperparameter set combination, or `None` if not doing grid search.

Return type (float, dict)

Raises

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

1.7.3 `config` Package

`skll.config.fix_json(json_string)`

Fixes incorrectly formatted quotes and capitalized booleans in the given JSON string.

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

Returns `json_string` – The normalized JSON string.

Return type `str`

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

Loads cross-validation folds from a CSV file with two columns for example ID and fold ID (and a header).

Parameters

- `folds_file` (*str*) – The path to a folds file to read.
- `ids_to_floats` (*bool, optional*) – Whether to convert IDs to floats. Defaults to `False`.

Returns `res` – A dictionary with example IDs as the keys and fold IDs as the values.

Return type `dict`

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

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

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

Parameters

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

Returns `path_to_check` – The normalized absolute path, if it exists.

Return type `str`

Raises `IOError` – If the file does not exist.

1.7.4 data Package

data.featureset Module

Classes related to storing/merging feature sets.

author Dan Blanchard (dblanchard@ets.org)

author Nitin Madnani (nmadnani@ets.org)

author Jeremy Biggs (jbiggs@ets.org)

organization ETS

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

Bases: `object`

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

Parameters

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

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

Notes

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

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

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

Parameters

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

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

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

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

Parameters

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

Yields

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

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

static from_data_frame (*df*, *name*, *labels_column=None*, *vectorizer=None*)
 Helper function to create a `FeatureSet` instance from a `pandas.DataFrame`. Will raise an Exception if pandas is not installed in your environment. The `ids` in the `FeatureSet` will be the index from the given frame.

Parameters

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

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

Return type `skll.FeatureSet`

`has_labels`

Check if `FeatureSet` has finite labels.

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

Return type `bool`

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

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

Parameters

- **fs** (`skll.FeatureSet`) – The `FeatureSet` instance to split.
- **ids_for_split1** (*list of int*) – A list of example IDs which will be split out into the first `FeatureSet` instance. Note that the `FeatureSet` instance will respect the order of the specified IDs.

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

Returns

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

data.readers Module

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

Notes about IDs & Label Conversion

All `Reader` sub-classes are designed to read in example IDs as strings unless `ids_to_floats` is set to `True` in which case they will be read in as floats, if possible. In the latter case, an exception will be raised if they cannot be converted to floats.

All `Reader` sub-classes also use the `safe_float` function internally to read in labels. This function tries to convert a single label first to `int`, then to `float`. If neither conversion is possible, the label remains a `str`. It should be noted that, if classification is being done with a feature set that is read in with one of the `Reader` sub-classes, care must be taken to ensure that labels do not get converted in unexpected ways. For example, classification labels should not be a mixture of `int`-converting and `float`-converting labels. Consider the situation below:

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

The labels will all be converted to floats and any classification model generated with this data will predict labels such as `2.0`, `2.2`, etc., not `str` values that exactly match the input labels, as might be expected. Be aware that it may be best to make use of the `class_map` keyword argument in such cases to map original labels to labels that convert only to `str`.

author Dan Blanchard (dblanchard@ets.org)

author Michael Heilman (mheilman@ets.org)

author Nitin Madnani (nmadnani@ets.org)

author Jeremy Biggs (jbiggs@ets.org)

organization ETS

class `skll.data.readers.ARFFReader` (*path_or_list*, ***kwargs*)

Bases: `skll.data.readers.DelimitedReader`

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

Parameters

- **path_or_list** (*str*) – The path to the ARFF file.
- **kwargs** (*dict*, *optional*) – Other arguments to the Reader object.

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

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

Parameters

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

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

Bases: `skll.data.readers.Reader`

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

Parameters

- **path_or_list** (*str*) – The path to a comma-delimited file.
- **replace_blanks_with** (*value*, *dict*, or *None*, *optional*) – Specifies a new value with which to replace blank values. Options are

```

- value = A (numeric) value with which to replace
  ↳ blank values.
- dict = A dictionary specifying the replacement
  ↳ value for each column.
- None = Blank values will be left as blanks, and
  ↳ not replaced.
    
```

The replacement occurs after the data set is read into a *pd.DataFrame*. Defaults to None.

- **drop_blanks** (*bool, optional*) – If True, remove lines/rows that have any blank values. These lines/rows are removed after the the data set is read into a *pd.DataFrame*. Defaults to False.
- **pandas_kwargs** (*dict or None, optional*) – Arguments that will be passed directly to the *pandas* I/O reader. Defaults to None.
- **kwargs** (*dict, optional*) – Other arguments to the Reader object.

```

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

Reader for creating a *FeatureSet* instance from a delimited (CSV/TSV) file. If example/instance IDs are included in the files, they must be specified in the *id* column. For ARFF, CSV, and TSV files, there must be a column with the name specified by *label_col* if the data is labeled. For ARFF files, this column must also be the final one (as it is in Weka).

Parameters

- **path_or_list** (*str*) – The path to a delimited file.
- **dialect** (*str*) – The dialect of to pass on to the underlying CSV reader. Defaults to 'excel-tab'.
- **kwargs** (*dict, optional*) – Other arguments to the Reader object.

```

class skll.data.readers.DictListReader (path_or_list,          quiet=True,
                                         ids_to_floats=False,    la-
                                         bel_col='y',            id_col='id',
                                         class_map=None,         sparse=True,
                                         feature_hasher=False,
                                         num_features=None,     log-
                                         ger=None)
    
```

Bases: *skll.data.readers.Reader*

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

read()

Read examples from list of dictionaries.

Returns `feature_set` – FeatureSet representing the list of dictionaries we read in.

Return type *skll.FeatureSet*

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

Bases: *skll.data.readers.Reader*

Reader to create a FeatureSet instance from a LibSVM/LibLinear/SVMLight file. We use a specially formatted comment for storing example IDs, class names, and feature names, which are normally not supported by the format. The comment is not mandatory, but without it, your labels and features will not have names. The comment is structured as follows:

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

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

Bases: *skll.data.readers.Reader*

Reader to create a FeatureSet instance from a JSONlines/NDJ file. If example/instance IDs are included in the files, they must be specified as the “id” key in each JSON dictionary.

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

Bases: object

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

Parameters

- **path_or_list** (*str or list of dict*) – Path or a list of example dictionaries.
- **quiet** (*bool, optional*) – Do not print “Loading…” status message to stderr. Defaults to True.
- **ids_to_floats** (*bool, optional*) – Convert IDs to float to save memory. Will raise error if we encounter an a non-numeric ID. Defaults to False.

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

classmethod for_path (*path_or_list, **kwargs*)

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

Parameters

- **path_or_list** (*str or list of dicts*) – A path or list of example dictionaries.
- **kwargs** (*dict, optional*) – The arguments to the Reader object being instantiated.

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

Return type `skll.Reader`

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

read ()

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

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

Return type `skll.FeatureSet`

Raises

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

```
class skll.data.readers.TSVReader (path_or_list, re-
                                     place_blanks_with=None,
                                     drop_blanks=False, pan-
                                     das_kwargs=None, **kwargs)
```

Bases: `skll.data.readers.CSVReader`

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

Parameters

- **`path_or_list`** (*str*) – The path to a comma-delimited file.
- **`replace_blanks_with`** (*value, dict, or None, optional*) – Specifies a new value with which to replace blank values. Options are

```
- value = A (numeric) value with which to replace_
  ↳ blank values.
- dict = A dictionary specifying the replacement_
  ↳ value for each column.
- None = Blank values will be left as blanks, and_
  ↳ not replaced.
```

The replacement occurs after the data set is read into a `pd.DataFrame`. Defaults to `None`.

- **`drop_blanks`** (*bool, optional*) – If `True`, remove lines/rows that have any blank values. These lines/rows are removed after the the data set is read into a `pd.DataFrame`. Defaults to `False`.
- **`pandas_kwargs`** (*dict or None, optional*) – Arguments that will be passed directly to the `pandas` I/O reader. Defaults to `None`.
- **`kwargs`** (*dict, optional*) – Other arguments to the Reader object.

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

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

Parameters

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

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

Return type int or float or str

data.writers Module

Handles loading data from various types of data files.

author Dan Blanchard (dblanchard@ets.org)

author Michael Heilman (mheilman@ets.org)

author Nitin Madnani (nmadnani@ets.org)

author Jeremy Biggs (jbiggs@ets.org)

organization ETS

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

Bases: `skll.data.writers.Writer`

Writer for writing out FeatureSets as ARFF files.

Parameters

- **path** (*str*) – A path to the feature file we would like to create. If `subsets` is not None, this is assumed to be a string containing the path to the directory to write the feature files with an additional file extension specifying the file type. For example `/foo/.arff`.
- **feature_set** (`skll.FeatureSet`) – The FeatureSet instance to dump to the output file.
- **relation** (*str, optional*) – The name of the relation in the ARFF file. Defaults to `'skll_relation'`.
- **regression** (*bool, optional*) – Is this an ARFF file to be used for regression? Defaults to `False`.

- **kwargs** (*dict, optional*) – The arguments to the `Writer` object being instantiated.

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

Bases: `skll.data.writers.Writer`

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

Parameters

- **path** (*str*) – A path to the feature file we would like to create. If `subsets` is not `None`, this is assumed to be a string containing the path to the directory to write the feature files with an additional file extension specifying the file type. For example `/foo/.csv`.
- **feature_set** (`skll.FeatureSet`) – The `FeatureSet` instance to dump to the output file.
- **pandas_kwargs** (*dict or None, optional*) – Arguments that will be passed directly to the `pandas` I/O reader. Defaults to `None`.
- **kwargs** (*dict, optional*) – The arguments to the `Writer` object being instantiated.

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

Bases: `skll.data.writers.Writer`

Writer for writing out `FeatureSets` as LibSVM/SVMLight files.

Parameters

- **path** (*str*) – A path to the feature file we would like to create. If `subsets` is not `None`, this is assumed to be a string containing the path to the directory to write the feature files with an additional file extension specifying the file type. For example `/foo/.libsvm`.
- **feature_set** (`skll.FeatureSet`) – The `FeatureSet` instance to dump to the output file.
- **kwargs** (*dict, optional*) – The arguments to the `Writer` object being instantiated.

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

Bases: `skll.data.writers.Writer`

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

Parameters

- **path** (*str*) – A path to the feature file we would like to create. If `subsets` is not `None`, this is assumed to be a string containing the path

to the directory to write the feature files with an additional file extension specifying the file type. For example `/foo/.ndj`.

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

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

Bases: `skll.data.writers.CSVWriter`

Writer for writing out FeatureSets as TSV files.

Parameters

- **path** (*str*) – A path to the feature file we would like to create. If `subsets` is not `None`, this is assumed to be a string containing the path to the directory to write the feature files with an additional file extension specifying the file type. For example `/foo/.tsv`.
- **feature_set** (`skll.FeatureSet`) – The `FeatureSet` instance to dump to the output file.
- **pandas_kwargs** (*dict or None, optional*) – Arguments that will be passed directly to the `pandas` I/O reader. Defaults to `None`.
- **kwargs** (*dict, optional*) – The arguments to the `Writer` object being instantiated.

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

Bases: `object`

Helper class for writing out FeatureSets to files on disk.

Parameters

- **path** (*str*) – A path to the feature file we would like to create. The suffix to this filename must be `.arff`, `.csv`, `.jsonlines`, `.libsvm`, `.ndj`, or `.tsv`. If `subsets` is not `None`, when calling the `write()` method, `path` is assumed to be a string containing the path to the directory to write the feature files with an additional file extension specifying the file type. For example `/foo/.csv`.
- **feature_set** (`skll.FeatureSet`) – The `FeatureSet` instance to dump to the file.
- **quiet** (*bool*) – Do not print “Writing...” status message to `stderr`. Defaults to `True`.
- **subsets** (*dict (str to list of str)*) – A mapping from subset names to lists of feature names that are included in those sets. If

given, a feature file will be written for every subset (with the name containing the subset name as suffix to `path`). Note, since string-valued features are automatically converted into boolean features with names of the form `FEATURE_NAME=STRING_VALUE`, when doing the filtering, the portion before the `=` is all that's used for matching. Therefore, you do not need to enumerate all of these boolean feature names in your mapping. Defaults to `None`.

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

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

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

Parameters

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

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

Return type *skll.data.writers.Writer*

write()

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

1.7.5 experiments Package

Functions for running and interacting with SKLL experiments.

author Nitin Madnani (nmadnani@ets.org)

author Dan Blanchard (dblanchard@ets.org)

author Michael Heilman (mheilman@ets.org)

author Chee Wee Leong (cleong@ets.org)

```
skll.experiments.generate_learning_curve_plots(experiment_name,  
                                              output_dir, learning_curve_tsv_file)
```

Generate the learning curve plots given the TSV output file from a learning curve experiment.

Parameters

- **experiment_name** (*str*) – The name of the experiment.
- **output_dir** (*str*) – Path to the output directory for the plots.
- **learning_curve_tsv_file** (*str*) – The path to the learning curve TSV file.

```
skll.experiments.load_featureset(dir_path, feat_files, suffix, id_col='id',  
                               label_col='y', ids_to_floats=False,  
                               quiet=False, class_map=None, feature_hasher=False,  
                               num_features=None, logger=None)
```

Load a list of feature files and merge them.

Parameters

- **dir_path** (*str*) – Path to the directory that contains the feature files.
- **feat_files** (*list of str*) – A list of feature file prefixes.
- **suffix** (*str*) – The suffix to add to feature file prefixes to get the full filenames.
- **id_col** (*str, optional*) – Name of the column which contains the instance IDs. If no column with that name exists, or *None* is specified, example IDs will be automatically generated. Defaults to 'id'.
- **label_col** (*str, optional*) – Name of the column which contains the class labels. If no column with that name exists, or *None* is specified, the data is considered to be unlabeled. Defaults to 'y'.
- **ids_to_floats** (*bool, optional*) – Whether to convert the IDs to floats to save memory. Will raise error if we encounter non-numeric IDs. Defaults to *False*.
- **quiet** (*bool, optional*) – Do not print “Loading...” status message to stderr. Defaults to *False*.
- **class_map** (*dict, optional*) – Mapping from original class labels to new ones. This is mainly used for collapsing multiple labels into a single class. Anything not in the mapping will be kept the same. Defaults to *None*.
- **feature_hasher** (*bool, optional*) – Should we use a Feature-Hasher when vectorizing features? Defaults to *False*.

- **num_features** (*int, optional*) – The number of features to use with the `FeatureHasher`. This should always be set to the power of 2 greater than the actual number of features you’re using. Defaults to `None`.
- **logger** (*logging.Logger, optional*) – A logger instance to use to log messages instead of creating a new one by default. Defaults to `None`.

Returns `merged_set` – A `FeatureSet` instance containing the specified labels, IDs, features, and feature vectorizer.

Return type *skll.FeatureSet*

```
skll.experiments.run_configuration(config_file, local=False, over-  
write=True, queue='all.q',  
hosts=None, write_summary=True,  
quiet=False, ablation=0, re-  
sume=False, log_level=20)
```

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

Parameters

- **config_file** (*str*) – Path to the configuration file we would like to use.
- **local** (*bool, optional*) – Should this be run locally instead of on the cluster? Defaults to `False`.
- **overwrite** (*bool, optional*) – If the model files already exist, should we overwrite them instead of re-using them? Defaults to `True`.
- **queue** (*str, optional*) – The DRMAA queue to use if we’re running on the cluster. Defaults to `'all.q'`.
- **hosts** (*list of str, optional*) – If running on the cluster, these are the machines we should use. Defaults to `None`.
- **write_summary** (*bool, optional*) – Write a TSV file with a summary of the results. Defaults to `True`.
- **quiet** (*bool, optional*) – Suppress printing of “Loading…” messages. Defaults to `False`.
- **ablation** (*int, optional*) – Number of features to remove when doing an ablation experiment. If positive, we will perform repeated ablation runs for all combinations of features removing the specified number at a time. If `None`, we will use all combinations of all lengths. If 0, the default, no ablation is performed. If negative, a `ValueError` is raised. Defaults to 0.

- **resume** (*bool, optional*) – If result files already exist for an experiment, do not overwrite them. This is very useful when doing a large ablation experiment and part of it crashes. Defaults to `False`.
- **log_level** (*str, optional*) – The level for logging messages. Defaults to `logging.INFO`.

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

Return type list of str

Raises

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

1.7.6 learner Package

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

author Nitin Madnani (nmadnani@ets.org)

author Michael Heilman (mheilman@ets.org)

author Dan Blanchard (dblanchard@ets.org)

author Aoife Cahill (acahill@ets.org)

organization ETS

```
class skill.learner.Learner(model_type, probability=False, pipeline=False,
                           feature_scaling='none', model_kwargs=None,
                           pos_label_str=None, min_feature_count=1,
                           sampler=None, sampler_kwargs=None, cus-
                           tom_learner_path=None, logger=None)
```

Bases: `object`

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

Parameters

- **model_type** (*str*) – Name of estimator to create (e.g., 'LogisticRegression'). See the skill package documentation for valid options.
- **probability** (*bool, optional*) – Should learner return probabilities of all labels (instead of just label with highest probability)? Defaults to `False`.

- **pipeline** (*bool, optional*) – Should learner contain a pipeline attribute that contains a scikit-learn Pipeline object composed of all steps including the vectorizer, the feature selector, the sampler, the feature scaler, and the actual estimator. Note that this will increase the size of the learner object in memory and also when it is saved to disk. Defaults to `False`.
- **feature_scaling** (*str, optional*) – How to scale the features, if at all. Options are - ‘with_std’: scale features using the standard deviation - ‘with_mean’: center features using the mean - ‘both’: do both scaling as well as centering - ‘none’: do neither scaling nor centering Defaults to ‘none’.
- **model_kwargs** (*dict, optional*) – A dictionary of keyword arguments to pass to the initializer for the specified model. Defaults to `None`.
- **pos_label_str** (*str, optional*) – A string denoting the label of the class to be treated as the positive class in a binary classification setting. If `None`, the class represented by the label that appears second when sorted is chosen as the positive class. For example, if the two labels in data are “A” and “B” and `pos_label_str` is not specified, “B” will be chosen as the positive class. Defaults to `None`.
- **min_feature_count** (*int, optional*) – The minimum number of examples a feature must have a nonzero value in to be included. Defaults to 1.
- **sampler** (*str, optional*) – The sampler to use for kernel approximation, if desired. Valid values are - ‘AdditiveChi2Sampler’ - ‘Nystroem’ - ‘RBFSampler’ - ‘SkewedChi2Sampler’ Defaults to `None`.
- **sampler_kwargs** (*dict, optional*) – A dictionary of keyword arguments to pass to the initializer for the specified sampler. Defaults to `None`.
- **custom_learner_path** (*str, optional*) – Path to module where a custom classifier is defined. Defaults to `None`.
- **logger** (*logging object, optional*) – A logging object. If `None` is passed, get logger from `__name__`. Defaults to `None`.

```
cross_validate (examples, stratified=True, cv_folds=10,
                grid_search=True, grid_search_folds=3, grid_jobs=None,
                grid_objective=None, output_metrics=[], pre-
                diction_prefix=None, param_grid=None, shuf-
                fle=False, save_cv_folds=False, save_cv_models=False,
                use_custom_folds_for_grid_search=True)
```

Cross-validates a given model on the training examples.

Parameters

- **examples** (`skll.FeatureSet`) – The `FeatureSet` instance to cross-validate learner performance on.
- **stratified** (`bool, optional`) – Should we stratify the folds to ensure an even distribution of labels for each fold? Defaults to `True`.
- **cv_folds** (`int or dict, optional`) – The number of folds to use for cross-validation, or a mapping from example IDs to folds. Defaults to 10.
- **grid_search** (`bool, optional`) – Should we do grid search when training each fold? Note: This will make this take *much* longer. Defaults to `False`.
- **grid_search_folds** (`int or dict, optional`) – The number of folds to use when doing the grid search, or a mapping from example IDs to folds. Defaults to 3.
- **grid_jobs** (`int, optional`) – The number of jobs to run in parallel when doing the grid search. If `None` or 0, the number of grid search folds will be used. Defaults to `None`.
- **grid_objective** (`str, optional`) – The name of the objective function to use when doing the grid search. Must be specified if `grid_search` is `True`. Defaults to `None`.
- **output_metrics** (`list of str, optional`) – List of additional metric names to compute in addition to the metric used for grid search. Empty by default. Defaults to an empty list.
- **prediction_prefix** (`str, optional`) – If saving the predictions, this is the prefix that will be used for the filename. It will be followed by `"_predictions.tsv"` Defaults to `None`.
- **param_grid** (`dict, optional`) – The parameter grid to search. Defaults to `None`.
- **shuffle** (`bool, optional`) – Shuffle examples before splitting into folds for CV. Defaults to `False`.
- **save_cv_folds** (`bool, optional`) – Whether to save the cv fold ids or not? Defaults to `False`.
- **save_cv_models** (`bool, optional`) – Whether to save the cv models or not? Defaults to `False`.
- **use_custom_folds_for_grid_search** (`bool, optional`) – If `cv_folds` is a custom dictionary, but `grid_search_folds` is not, perhaps due to user oversight,

should the same custom dictionary automatically be used for the inner grid-search cross-validation? Defaults to `True`.

Returns

- **results** (*list of 6-tuples*) – The confusion matrix, overall accuracy, per-label PRFs, model parameters, objective function score, and evaluation metrics (if any) for each fold.
- **grid_search_scores** (*list of floats*) – The grid search scores for each fold.
- **grid_search_cv_results_dicts** (*list of dicts*) – A list of dictionaries of grid search CV results, one per fold, with keys such as “params”, “mean_test_score”, etc, that are mapped to lists of values associated with each hyperparameter set combination.
- **skll_fold_ids** (*dict*) – A dictionary containing the test-fold number for each id if `save_cv_folds` is `True`, otherwise `None`.
- **models** (*list of skll.learner.Learner*) – A list of `skll.learner.Learner`s, one for each fold if `save_cv_models` is `True`, otherwise `None`.

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

evaluate (*examples*, *prediction_prefix=None*, *append=False*,
grid_objective=None, *output_metrics=[]*)
 Evaluates a given model on a given dev or test `FeatureSet`.

Parameters

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

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

additional evaluation metrics, if any. For regressors, the first two elements are None.

Return type 6-tuple

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

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

Parameters

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

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

Return type *skll.Learner*

Raises

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

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

Generates learning curves for a given model on the training examples via cross-validation. Adapted from the scikit-learn code for learning curve generation (cf. “`sklearn.model_selection.learning_curve`”).

Parameters

- **examples** (*skll.FeatureSet*) – The `FeatureSet` instance to generate the learning curve on.
- **cv_folds** (*int or dict, optional*) – The number of folds to use for cross-validation, or a mapping from example IDs to folds. Defaults to 10.
- **metric** (*str*) – The name of the metric function to use when computing the train and test scores for the learning curve.
- **train_sizes** (*list of float or int, optional*) – Relative or absolute numbers of training examples that will be used to generate the learning curve. If the type is float, it is regarded as a fraction of the maximum size of the training set (that is determined by the selected validation method), i.e. it has to be within (0, 1]. Otherwise it is interpreted as absolute sizes of the training sets. Note that for classification the number of samples usually have to be big enough to contain at least one sample from each class. Defaults to `np.linspace(0.1, 1.0, 5)`.

Returns

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

load (*learner_path*)

Replace the current learner instance with a saved learner.

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

model

The underlying scikit-learn model

model_kwargs

A dictionary of the underlying scikit-learn model’s keyword arguments

model_params

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

Returns

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

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

model_type

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

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

Uses a given model to return, and optionally, write out predictions on a given `FeatureSet` to a file.

For regressors, the returned and written-out predictions are identical. However, for classifiers: - if `class_labels` is `True`, class labels are returned

as well as written out.

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

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

Parameters

- **examples** (`skll.FeatureSet`) – The `FeatureSet` instance to predict labels for.
- **prediction_prefix** (`str`, *optional*) – If not `None`, predictions will also be written out to a file with the name `<prediction_prefix>_predictions.tsv`. For classifiers, the predictions written out are class labels unless the learner is probabilistic AND `class_labels` is set to `False`. Note that this prefix can also contain a path. Defaults to `None`.
- **append** (`bool`, *optional*) – Should we append the current predictions to the file if it exists? Defaults to `False`.
- **class_labels** (`bool`, *optional*) – If `False`, return either the class probabilities (probabilistic classifiers) or the class indices (non-probabilistic ones). If `True`, return the class labels no matter what. Ignored for regressors. Defaults to `True`.

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

Return type array-like

Raises

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

probability

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

save (`learner_path`)

Save the `Learner` instance to a file.

Parameters `learner_path` (`str`) – The path to save the `Learner` instance to.

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

Train the model underlying the learner and return the grid search score and a dictionary of grid search results.

Parameters

- **examples** (`skll.FeatureSet`) – The `FeatureSet` instance to use for training.
- **param_grid** (`dict`, *optional*) – The parameter grid to search through for grid search. If `None`, a default parameter grid will be used. Defaults to `None`.
- **grid_search_folds** (`int` or `dict`, *optional*) – The number of folds to use when doing the grid search, or a mapping from example IDs to folds. Defaults to 3.
- **grid_search** (`bool`, *optional*) – Should we do grid search? Defaults to `True`.
- **grid_objective** (`str`, *optional*) – The name of the objective function to use when doing the grid search. Must be specified if `grid_search` is `True`. Defaults to `None`.
- **grid_jobs** (`int`, *optional*) – The number of jobs to run in parallel when doing the grid search. If `None` or 0, the number of grid search folds will be used. Defaults to `None`.
- **shuffle** (`bool`, *optional*) – Shuffle examples (e.g., for grid search CV.) Defaults to `False`.

Returns tuple – 1) The best grid search objective function score, or 0 if we’re not doing grid search, and 2) a dictionary of grid search CV results with keys such as “params”, “mean_test_score”, etc, that are mapped to lists of values associated with each hyperparameter set combination, or `None` if not doing grid search.

Return type (float, dict)

Raises

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

`skll.learner.load_custom_learner` (*custom_learner_path*, *cus-
tom_learner_name*)

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

Parameters

- **custom_learner_path** (`str`) – The path to a custom learner.

- **custom_learner_name** (*str*) – The name of a custom learner.

Raises `ValueError` – If the custom learner path does not end in `‘.py’`.

Returns **custom_learner_obj** – The SKLL learner object loaded from the given path.

Return type `skll.Learner` object

1.7.7 metrics Module

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

author Nitin Madnani (nmadnani@ets.org)

author Michael Heilman (mheilman@ets.org)

author Dan Blanchard (dblanchard@ets.org)

organization ETS

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

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

Parameters

- **y_true** (*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.
- **corr_type** (*str, optional*) – Which type of correlation to compute. Possible choices are `pearson`, `spearman`, and `kendall_tau`. Defaults to `pearson`.

Returns **ret_score** – correlation value if well-defined, else 0.0

Return type `float`

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

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

Parameters

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

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

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

Return type float

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

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

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

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

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

Parameters

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

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

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

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

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

Return type float

Raises

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

`skll.metrics.register_custom_metric` (*custom_metric_path*, *cus-*
tom_metric_name)

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

Parameters

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

Raises

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

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

Call the scoring function in `sklearn.metrics.SCORERS` with the given name. This takes care of handling keyword arguments that were pre-specified when creating the scorer. This applies any sign-flipping that was specified by `make_scorer()` when the scorer was created.

Parameters

- **`func_name`** (*str*) – The name of the objective function to use from `SCORERS`.
- **`y_true`** (*array-like of float*) – The true/actual/gold labels for the data.
- **`y_pred`** (*array-like of float*) – The predicted/observed labels for the data.

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

Return type float

1.7.8 `utils` Package

Various useful constants defining groups of evaluation metrics.

`skll.utils.constants.CLASSIFICATION_ONLY_METRICS = {'accuracy', 'average_precision'}`
 Set of evaluation metrics only used for classification tasks

`skll.utils.constants.CORRELATION_METRICS = {'kendall_tau', 'pearson', 'spearman'}`
 Set of evaluation metrics based on correlation

`skll.utils.constants.PROBABILISTIC_METRICS = frozenset({'neg_log_loss', 'roc_auc'})`
 Set of evaluation metrics that can use prediction probabilities

`skll.utils.constants.REGRESSION_ONLY_METRICS = {'explained_variance', 'max_error'}`
 Set of evaluation metrics only used for regression tasks

`skll.utils.constants.UNWEIGHTED_KAPPA_METRICS = {'unweighted_kappa', 'uwkappa'}`
 Set of unweighted kappa agreement metrics

`skll.utils.constants.WEIGHTED_KAPPA_METRICS = {'linear_weighted_kappa', 'lwkappa'}`
 Set of weighed kappa agreement metrics

A useful logging function for SKLL developers

`skll.utils.logging.get_skll_logger(name, filepath=None, log_level=20)`
 Create and return logger instances that are appropriate for use in SKLL code, e.g., that they can log to both `STDERR` as well as a file. This function will try to reuse any previously created logger based on the given name and filepath.

Parameters

- **name** (*str*) – The name to be used for the logger.
- **filepath** (*str, optional*) – The file to be used for the logger via a `FileHandler`. Default: `None` in which case no file is attached to the logger. Defaults to `None`.
- **log_level** (*str, optional*) – The level for logging messages Defaults to `logging.INFO`.

Returns `logger` – A `Logger` instance.

Return type `logging.Logger`

1.8 Contributing

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

1.8.1 Guidelines

The SKLL contribution guidelines can be found in our Github repository [here](#). Please try to follow them as much as possible.

1.8.2 SKLL Code Overview

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

Organization

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

- `__init__.py` : Code used to initialize the `skll` Python package.
- `config.py` : Code to parse SKLL experiment configuration files.
- `experiments.py` : Code that is related to creating and running SKLL experiments. It also contains code that collects the various evaluation metrics and predictions for each SKLL experiment and writes them out to disk.
- `learner.py` : Code for the `Learner` class. This class is instantiated for every learner name specified in the experiment configuration file.
- `metrics.py` : Code for any custom metrics that are not in `sklearn.metrics`, e.g., `kappa`, `kendall_tau`, `spearman`, etc.
- `logutils.py` : Code for a custom logging solution that allows capturing any information logged to `STDOUT/STDERR` by SKLL and `scikit-learn` to also be captured into log files that are then saved on disk.
- `version.py` : Code to define the SKLL version. Only changed for new releases.
- `data/`
 - `__init__.py` : Code used to initialize the `skll.data` Python package.
 - `featureset.py` : Code for the `FeatureSet` class – the main class that encapsulates all of the features names, values, and metadata for a given set of instances.
 - `readers.py` : Code for classes that can read various file formats and create `FeatureSet` objects from them.
 - `writers.py` : Code for classes that can write `FeatureSet` objects to files on disk in various formats.
 - `dict_vectorizer.py` : Code for a `DictVectorizer` class that subclasses `sklearn.feature_extraction.DictVectorizer` to add an `__eq__()` method that we need for vectorizer equality.
- `utilities/`
 - `compute_eval_from_predictions.py` : See documentation.

- `filter_features.py` : See documentation.
- `generate_predictions.py` : See documentation.
- `join_features.py` : See documentation.
- `plot_learning_curves.py` : See documentation.
- `print_model_weights.py` : See documentation.
- `run_experiment.py` : See documentation.
- `skll_convert.py` : See documentation.
- `summarize_results.py` : See documentation.
- `tests/`
 - `test_*.py` : These files contain the code for the unit tests and regression tests.

Entry Points & Workflow

There are three main entry points into the SKLL codebase:

1. **Experiment configuration files.** The primary way to interact with SKLL is by writing configuration files and then passing it to the `run_experiment` script. When you run the command `run_experiment <config_file>`, the following happens (at a high level):
 - the configuration file is handed off to the `run_configuration()` function in `experiments.py`.
 - a `SKLLConfigParser` object is instantiated from `config.py` that parses all of the relevant fields out of the given configuration file.
 - the configuration fields are then passed to the `_classify_featureset()` function in `experiments.py` which instantiates the learners (using code from `learner.py`), the featuresets (using code from `reader.py` & `featureset.py`), and runs the experiments, collects the results, and writes them out to disk.
2. **SKLL API.** Another way to interact with SKLL is via the SKLL API directly in your Python code rather than using configuration files. For example, you could use the `Learner.from_file()` method to load a saved model from disk and make predictions on new data. The documentation for the SKLL API can be found [here](#).
3. **Utility scripts.** The scripts listed in the section above under `utilities` are also entry points into the SKLL code. These scripts are convenient wrappers that use the SKLL API for commonly used tasks, e.g., generating predictions on new data from an already trained model.

1.9 Internal Documentation

1.9.1 Release Process

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

1. Create a release branch `release/XX` on GitHub.
2. In the release branch:
 - a. update the version numbers in `version.py`.
 - b. update the conda recipe.
 - c. update the documentation with any new features or details about changes.
 - d. run `make linkcheck` on the documentation and fix any redirected/broken links.
 - e. update the README and this release documentation, if necessary.
3. Run the following command in the `conda-recipe` directory to build the conda package:

```
conda build -c conda-forge --numpy=1.17 .
```

4. Upload the package to `anaconda.org` using `anaconda upload --user ets <package tarball>`. You will need to have the appropriate permissions for the `ets` organization.
5. Build the PyPI source distribution using `python setup.py sdist build`.
6. Upload the source distribution to TestPyPI using `twine upload --repository testpypi dist/*`. You will need to have the `twine` package installed and set up your `$HOME/.pypirc` correctly. See details [here](#).
7. Test the conda package by creating a new environment on different platforms with this package installed and then running SKLL examples or tests from a SKLL working copy. If the package works, then move on to the next step. If it doesn't, figure out why and rebuild and re-upload the package.
8. Test the TestPyPI package by installing it as follows:

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

9. Then run some SKLL examples or tests from a SKLL working copy. If the TestPyPI package works, then move on to the next step. If it doesn't, figure out why and rebuild and re-upload the package.
10. Create pull requests on the [skll-conda-tester](#) and [skll-pip-tester](#) repositories to test the conda and TestPyPI packages on Linux and Windows.

11. Draft a release on GitHub while the Linux and Windows package tester builds are running.
12. Once both builds have passed, make a pull request with the release branch to be merged into `master` and request code review.
13. Once the build for the PR passes and the reviewers approve, merge the release branch into `master`.
14. Upload source and wheel packages to PyPI using `python setup.py sdist upload` and `python setup.py bdist_wheel upload`
15. Make sure that the ReadTheDocs build for `master` passes.
16. Tag the latest commit in `master` with the appropriate release tag and publish the release on GitHub.
17. Send an email around at ETS announcing the release and the changes.
18. Post release announcement on Twitter/LinkedIn.

CHAPTER 2

Indices and tables

- `genindex`
- `modindex`
- `search`

S

`skll.data.featureset`, 61
`skll.data.readers`, 64
`skll.data.writers`, 70
`skll.experiments`, 73
`skll.learner`, 76
`skll.metrics`, 84

Symbols

-arff_regression
 skll_convert command line
 option, 46

-arff_relation ARFF_RELATION
 skll_convert command line
 option, 46

-id_col <id_col>
 filter_features command line
 option, 42

-k <k>
 print_model_weights command
 line option, 45

-no_labels
 skll_convert command line
 option, 46

-reuse_libsvm_map
 REUSE_LIBSVM_MAP
 skll_convert command line
 option, 46

-sign {positive,negative,all}
 print_model_weights command
 line option, 45

-sort_by_labels
 print_model_weights command
 line option, 46

-version
 compute_eval_from_predictions
 command line option, 42
 filter_features command line
 option, 43
 generate_predictions command
 line option, 44
 join_features command line
 option, 45
 print_model_weights command
 line option, 46
 run_experiment command line
 option, 35
 skll_convert command line
 option, 46
 summarize_results command
 line option, 47

-A, -ablation_all
 run_experiment command line
 option, 35

-I <id <id ...>>, -id <id <id
 ...>>
 filter_features command line
 option, 42

-L <label <label ...>>, -label
 <label <label ...>>
 filter_features command line
 option, 43

-a <num_features>, -ablation
 <num_features>
 run_experiment command line
 option, 34

-a, -ablation
 summarize_results command
 line option, 47

-db, -drop-blanks
 filter_features command line
 option, 43

```

-f <feature <feature ...>>,
  -feature <feature <feature
  ...>>
  filter_features command line
  option, 42
-i <id_col>, -id_col <id_col>
  generate_predictions command
  line option, 44
-i, -inverse
  filter_features command line
  option, 42
-k, -keep-models
  run_experiment command line
  option, 35
-l <label_col>, -label_col
  <label_col>
  filter_features command line
  option, 43
  generate_predictions command
  line option, 44
  join_features command line
  option, 45
  skll_convert command line
  option, 46
-l, -local
  run_experiment command line
  option, 35
-m <machines>, -machines
  <machines>
  run_experiment command line
  option, 35
-o <path>, -output_file <path>
  generate_predictions command
  line option, 44
-p, -predict_labels
  generate_predictions command
  line option, 44
-q <queue>, -queue <queue>
  run_experiment command line
  option, 35
-q, -quiet
  filter_features command line
  option, 43
  generate_predictions command
  line option, 44
  join_features command line
  option, 45
  skll_convert command line
  option, 46
-r, -resume
  run_experiment command line
  option, 35
-rb <replacement>,
  -replace-blanks-with
  <replacement>
  filter_features command line
  option, 43
-t <threshold>, -threshold
  <threshold>
  generate_predictions command
  line option, 44
-v, -verbose
  run_experiment command line
  option, 35

```

A

ARFFReader (*class in skll.data.readers*), 65

ARFFWriter (*class in skll.data.writers*), 70

C

CLASSIFICATION_ONLY_METRICS (*in module skll.utils.constants*), 86

compute_eval_from_predictions command line option

-version, 42

examples_file, 42

metric_names, 42

predictions_file, 42

correlation() (*in module skll.metrics*), 84

CORRELATION_METRICS (*in module skll.utils.constants*), 87

cross_validate() (*skll.Learner method*), 54

cross_validate() (*skll.learner.Learner method*), 77

CSVReader (*class in skll.data.readers*), 65

CSVWriter (*class in skll.data.writers*), 71

D

DelimitedReader (*class in*

`skll.data.readers`), 66
DictListReader (class in `skll.data.readers`), 66

E

`evaluate()` (*skll.Learner* method), 55
`evaluate()` (*skll.learner.Learner* method), 79
`examples_file`
 `compute_eval_from_predictions`
 command line option, 42

F

`f1_score_least_frequent()` (in module *skll.metrics*), 84
FeatureSet (class in *skll*), 48
FeatureSet (class in *skll.data.featureset*), 61
`filter()` (*skll.data.featureset.FeatureSet* method), 61
`filter()` (*skll.FeatureSet* method), 49
`filter_features` command line option
 option
 `-id_col <id_col>`, 42
 `-version`, 43
 `-I <id <id ...>>`, `-id <id <id ...>>`, 42
 `-L <label <label ...>>`, `-label <label <label ...>>`, 43
 `-db`, `-drop-blanks`, 43
 `-f <feature <feature ...>>`, `-feature <feature <feature ...>>`, 42
 `-i`, `-inverse`, 42
 `-l <label_col>`, `-label_col <label_col>`, 43
 `-q`, `-quiet`, 43
 `-rb <replacement>`, `-replace-blanks-with <replacement>`, 43
 `infile`, 42
 `outfile`, 42
`filtered_iter()`

(*skll.data.featureset.FeatureSet* method), 62

`filtered_iter()` (*skll.FeatureSet* method), 49
`fix_json()` (in module *skll.config*), 60
`for_path()` (*skll.data.readers.Reader* class method), 68
`for_path()` (*skll.data.writers.Writer* class method), 73
`from_data_frame()` (*skll.data.featureset.FeatureSet* static method), 63
`from_data_frame()` (*skll.FeatureSet* static method), 50
`from_file()` (*skll.Learner* class method), 56
`from_file()` (*skll.learner.Learner* class method), 80

G

`generate_learning_curve_plots()` (in module *skll.experiments*), 73
`generate_predictions` command line option
 `-version`, 44
 `-i <id_col>`, `-id_col <id_col>`, 44
 `-l <label_col>`, `-label_col <label_col>`, 44
 `-o <path>`, `-output_file <path>`, 44
 `-p`, `-predict_labels`, 44
 `-q`, `-quiet`, 44
 `-t <threshold>`, `-threshold <threshold>`, 44
 `input_file(s)`, 43
 `model_file`, 43
`get_skll_logger()` (in module *skll.utils.logging*), 87

H

`has_labels` (*skll.data.featureset.FeatureSet* attribute), 63
`has_labels` (*skll.FeatureSet* attribute), 51

I

`infile`

filter_features command line option, 42
 skll_convert command line option, 46
 infile ...
 join_features command line option, 44
 input_file(s)
 generate_predictions command line option, 43

J

join_features command line option
 -version, 45
 -l <label_col>, -label_col <label_col>, 45
 -q, -quiet, 45
 infile ..., 44
 outfile, 44
 json_file
 summarize_results command line option, 47

K

kappa() (in module *skll.metrics*), 85

L

Learner (class in *skll*), 52
 Learner (class in *skll.learner*), 76
 learning_curve() (*skll.Learner* method), 56
 learning_curve() (*skll.learner.Learner* method), 80
 LibSVMReader (class in *skll.data.readers*), 67
 LibSVMWriter (class in *skll.data.writers*), 71
 load() (*skll.Learner* method), 57
 load() (*skll.learner.Learner* method), 81
 load_custom_learner() (in module *skll.learner*), 83
 load_cv_folds() (in module *skll.config*), 60

load_featureset() (in module *skll.experiments*), 74
 locate_file() (in module *skll.config*), 60

M

metric_names
 compute_eval_from_predictions command line option, 42
 model (*skll.Learner* attribute), 57
 model (*skll.learner.Learner* attribute), 81
 model_file
 generate_predictions command line option, 43
 print_model_weights command line option, 45
 model_kwargs (*skll.Learner* attribute), 57
 model_kwargs (*skll.learner.Learner* attribute), 81
 model_params (*skll.Learner* attribute), 57
 model_params (*skll.learner.Learner* attribute), 81
 model_type (*skll.Learner* attribute), 57
 model_type (*skll.learner.Learner* attribute), 81

N

NDJReader (class in *skll.data.readers*), 67
 NDJWriter (class in *skll.data.writers*), 71

O

outfile
 filter_features command line option, 42
 join_features command line option, 44
 skll_convert command line option, 46
 output_dir
 plot_learning_curves command line option, 45

P

plot_learning_curves command line option
 output_dir, 45

tsv_file, 45
 predict() (*skll.Learner method*), 57
 predict() (*skll.learner.Learner method*), 81
 predictions_file
 compute_eval_from_predictions
 command line option, 42
 print_model_weights command
 line option
 -k <k>, 45
 -sign {positive,negative,all},
 45
 -sort_by_labels, 46
 -version, 46
 model_file, 45
 PROBABILISTIC_METRICS (*in module
 skll.utils.constants*), 87
 probability (*skll.Learner attribute*), 58
 probability (*skll.learner.Learner at-
 tribute*), 82

R

read() (*skll.data.readers.DictListReader
 method*), 66
 read() (*skll.data.readers.Reader method*), 68
 Reader (*class in skll.data.readers*), 67
 register_custom_metric() (*in module
 skll.metrics*), 86
 REGRESSION_ONLY_METRICS (*in module
 skll.utils.constants*), 87
 run_configuration() (*in module skll*),
 51
 run_configuration() (*in module
 skll.experiments*), 75
 run_experiment command line
 option
 -version, 35
 -A, -ablation_all, 35
 -a <num_features>, -ablation
 <num_features>, 34
 -k, -keep-models, 35
 -l, -local, 35
 -m <machines>, -machines
 <machines>, 35
 -q <queue>, -queue <queue>, 35
 -r, -resume, 35

-v, -verbose, 35

S

safe_float() (*in module skll.data.readers*),
 69
 save() (*skll.Learner method*), 59
 save() (*skll.learner.Learner method*), 82
 skll.data.featureset (*module*), 61
 skll.data.readers (*module*), 64
 skll.data.writers (*module*), 70
 skll.experiments (*module*), 73
 skll.learner (*module*), 76
 skll.metrics (*module*), 84
 skll_convert command line
 option
 -arff_regression, 46
 -arff_relation ARFF_RELATION,
 46
 -no_labels, 46
 -reuse_libsvm_map
 REUSE_LIBSVM_MAP, 46
 -version, 46
 -l <label_col>, -label_col
 <label_col>, 46
 -q, -quiet, 46
 infile, 46
 outfile, 46
 split_by_ids() (*skll.data.featureset.FeatureSet static
 method*), 63
 split_by_ids() (*skll.FeatureSet static
 method*), 51
 split_with_quotes() (*skll.data.readers.ARFFReader static
 method*), 65
 summarize_results command line
 option
 -version, 47
 -a, -ablation, 47
 json_file, 47
 summary_file, 47
 summary_file
 summarize_results command
 line option, 47

T

`train()` (*skll.Learner method*), 59
`train()` (*skll.learner.Learner method*), 82
`tsv_file`
 `plot_learning_curves` command
 line option, 45
`TSVReader` (*class in skll.data.readers*), 69
`TSVWriter` (*class in skll.data.writers*), 72

U

`UNWEIGHTED_KAPPA_METRICS` (*in module skll.utils.constants*), 87
`use_score_func()` (*in module skll.metrics*), 86

W

`WEIGHTED_KAPPA_METRICS` (*in module skll.utils.constants*), 87
`write()` (*skll.data.writers.Writer method*), 73
`Writer` (*class in skll.data.writers*), 72