mlens Documentation

Release 0.1.6

Sebastian Flennerhag

Nov 05, 2017

Installation

A Python library for memory efficient parallelized ensemble learning

NOTE: This site hosts documentation for version 0.1.6. Visit ml-ensemble.com for up-to-date documentation.

ML-Ensemble is a library for building Scikit-learn compatible ensemble estimator. By leveraging API elements from deep learning libraries like Keras for building ensembles, it is straightforward to build deep ensembles with complex interactions.

ML-Ensemble is open for contributions at all levels.If you would like to get involved, reach out to the project's Github repository. We are currently in beta testing, so please report any bugs or issues by creating an issue. If you are interested in contributing to development, see *Hacking ML-Ensemble* for a quick introduction to ensemble implementation, or check out the issue tracker.

CHAPTER 1

Core Features

1.1 Modular build of multi-layered ensembles

Ensembles are build as a feed-forward network, with a set of **layers** stacked on each other. Each layer is associated with a library of base learners, a mapping from preprocessing pipelines to subsets of base learners, and an estimation method. Layers are stacked sequentially with each layer taking the previous layer's output as input. You can propagate features through layers, differentiate preprocessing between subsets of base learners, vary the estimation method between layers and much more to build ensembles of almost any shape and form.

1.2 Transparent Architecture API

Ensembles are built by adding layers to an instance object: layers in their turn are comprised of a list of estimators. No matter how complex the ensemble, to train it call the `fit` method:

```
ensemble = Subsemble()

# First layer
ensemble.add(list_of_estimators)

# Second layer
ensemble.add(list_of_estimators)

# Final meta estimator
ensemble.add_meta(estimator)

# Train ensemble
ensemble.fit(X, y)
```

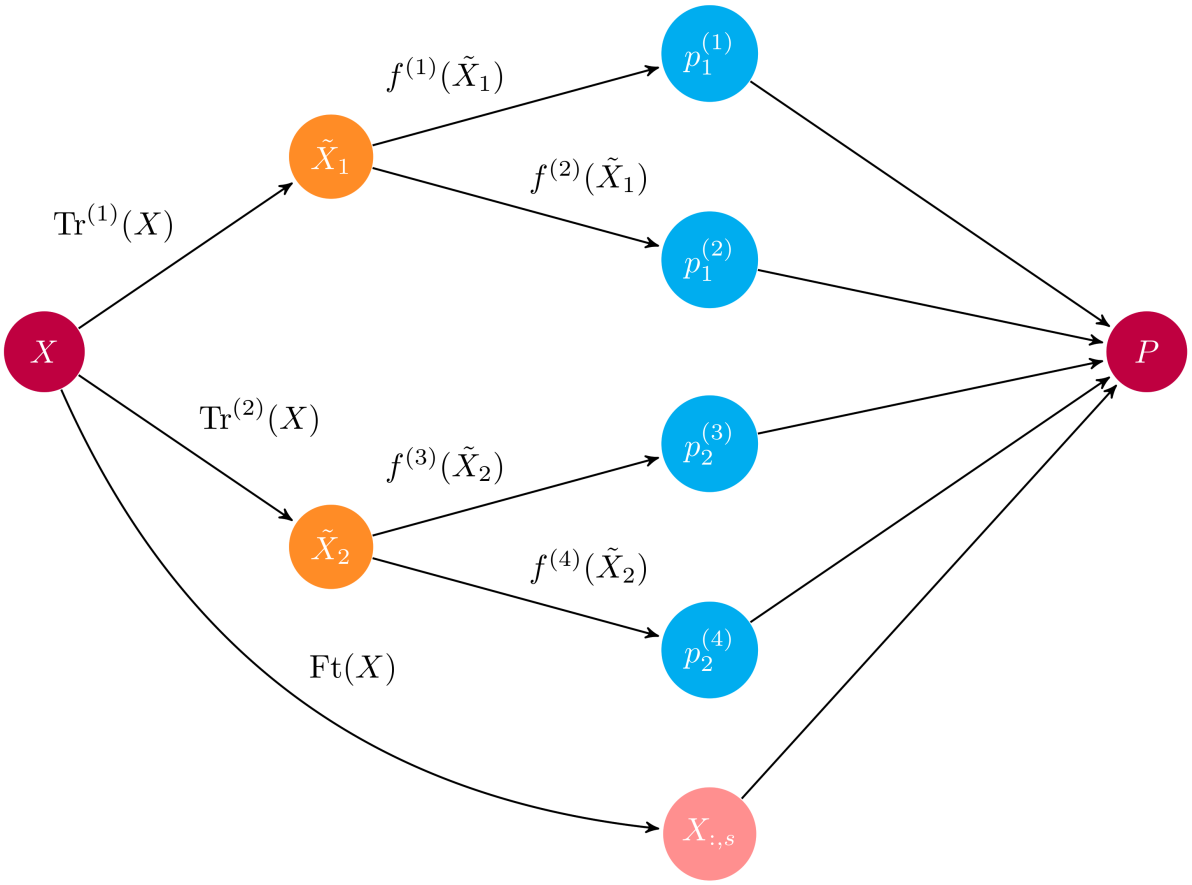


Fig. 1.1: The computational graph of a layer. The input X is either the original data or the previous layer's output; $\mathrm{Tr}^{(j)}$ represents preprocessing pipelines that transform the input to its associated base learners $f^{(i)}$. The Ft operation propagates specified features s from input to output. Base learner predictions $p_j^{(i)}$ are concatenated to propagated features $X_{:,s}$ to form the output matrix P.

1.3 Memory Efficient Parallelized Learning

Because base learners in an ensemble are independent of each other, ensembles benefit greatly from parallel processing. ML-Ensemble is designed to maximize parallelization at minimum memory footprint. By sharing memory, workers avoid transmitting and copying data between estimations. As such, ML-Ensemble typically require no more memory than sequential processing. For more details, see *Memory consumption*.

Expect 95-97% of training time to be spent fitting the base estimators. Training time depends primarily on the number of base learners in the ensemble, the number of threads or cores available, and the size of the dataset. Speaking of size, ensembles that partition the data during training scale more efficiently than their base learners.

1.4 Differentiated preprocessing pipelines

As mentioned, ML-Ensemble offers the possibility to specify for each layer a set of preprocessing pipelines to map to subsets (or all) of the layer's base learners. For instance, for one set of estimators, min-max-scaling might be desired, while for a different set of estimators standardization could be preferred.

```
ensemble = SuperLearner()

preprocessing = {'pipeline-1': list_of_transformers_1,
                 'pipeline-2': list_of_transformers_2}

estimators = {'pipeline-1': list_of_estimators_1,
              'pipeline-2': list_of_estimators_2}

ensemble.add(estimators, preprocessing)
```

1.5 Dedicated Diagnostics

To efficiently building complex ensembles, it is necessary to compare and contrast a variety of base learner set up. ML-Ensemble is equipped with a model selection suite that lets you compare several models across any number of preprocessing pipelines, all in one go. Ensemble transformers can be used to "preprocess" the input data according to how the initial layers of the ensemble would predict, to run cross-validated model selection on the ensemble output. Output is summarized for easy comparison of performance.

```
>>> DataFrame(evaluator.summary)
           test_score_mean  test_score_std  train_score_mean  train_score_std  fit_
↪time_mean  fit_time_std                                              params
class rf          0.955357        0.060950          0.972535         0.008303       0.
↪024585      0.014300                                    {'max_depth': 5}
      svc         0.961607        0.070818          0.972535         0.008303       0.
↪000800      0.000233                                  {'C': 7.67070164682}
proba rf          0.980357        0.046873          0.992254         0.007007       0.
↪022789      0.003296   {'max_depth': 3, 'max_features': 0.883535082341}
      svc         0.974107        0.051901          0.969718         0.008060       0.
↪000994      0.000367                                  {'C': 0.209602254061}
```

1.5.1 Install

ML-Ensemble is available through PyPi. For latest stable version, install `mlens` through `pip`.

```
pip install mlens
```

Bleeding edge

To latest stable development version can be install through the `master` branch of the *mlens* repository.

```
git clone https://github.com/flennerhag/mlens.git; cd mlens;
python install setup.py
```

Developer

The developmer version can be installed through the `dev` branch of the *mlens* repository. It is advised to check the CI build status first to ensure the branch builds correctly.

Dependencies

To install *mlens* the following dependencies are required:

Package	Version	Module
scipy	>= 0.17	All
numpy	>= 1.11	All

Additionally, to use the `visualization` module, the following libraries are necessary:

Package	Version
matplotlib	>= 1.5
seaborn	>= 0.7

If you want to run examples, you may also need:

Package	Version
sklearn	>= 0.17
pandas	>= 0.17

1.5.2 Test build

To test the installation, run:

```
cd mlens;
python check_build.py
```

Note that this requires the Nose unit testing suite: if not found, the test script will automatically try to install it using `pip install nose-exclude`. The expected output should look like:

```
>>> python check_build.py
Setting up tests... Ready.
Checking build... Build ok.
```

If the build fails, a log file will be created named `check_build_log.txt` that contains the traceback for the failed test for debugging.

1.5.3 Getting started

To get you up and running, the following guides highlights the basics of the API for ensemble classes, model selection and visualization.

Guides	Content
Ensemble guide	how to build, fit and predict with an ensemble
Model selection guide	how to compare several estimators in one go
Visualization guide	plotting functionality

For more more in-depth material and advanced usage, see *Tutorials*.

Preliminaries

We use the following setup throughout:

```
import numpy as np
from pandas import DataFrame
from sklearn.metrics import f1_score
from sklearn.datasets import load_iris

seed = 2017
np.random.seed(seed)

def f1(y, p): return f1_score(y, p, average='micro')

data = load_iris()
idx = np.random.permutation(150)
X = data.data[idx]
y = data.target[idx]
```

Ensemble guide

Building an ensemble

Instantiating a fully specified ensemble is straightforward and requires three steps: first create the instance, second add the intermediate layers, and finally the meta estimator.

```
from mlens.ensemble import SuperLearner
from sklearn.linear_model import LogisticRegression
from sklearn.ensemble import RandomForestClassifier
from sklearn.svm import SVC

# --- Build ---

# Passing a scoring function will create cv scores during fitting
# the scorer should be a simple function accepting to vectors and returning a scalar
ensemble = SuperLearner(scorer=f1, random_state=seed)

# Build the first layer
ensemble.add([RandomForestClassifier(random_state=seed), SVC()])

# Attach the final meta estimator
ensemble.add_meta(LogisticRegression())

# --- Use ---
```

```
# Fit ensemble
ensemble.fit(X[:75], y[:75])

# Predict
preds = ensemble.predict(X[75:])
```

To check the performance of estimator in the layers, call the `scores_` attribute. The attribute can be wrapped in a `pandas.DataFrame` for a tabular format.

```
>>> DataFrame(ensemble.scores_)
                                 score_mean  score_std
layer-1 randomforestclassifier     0.839260   0.055477
        svc                        0.894026   0.051920
```

To round off, let's see how the ensemble as a whole fared.

```
>>> f1(preds, y[75:])
0.95999999999999996
```

Multi-layer ensembles

With each call to the `add` method, another layer is added to the ensemble. Note that all ensembles are *sequential* in the order layers are added. For instance, in the above example, we could add a second layer as follows.

```
ensemble = SuperLearner(scorer=f1, random_state=seed, verbose=True)

# Build the first layer
ensemble.add([RandomForestClassifier(random_state=seed), LogisticRegression()])

# Build the second layer
ensemble.add([LogisticRegression(), SVC()])

# Attach the final meta estimator
ensemble.add_meta(SVC())
```

We now fit this ensemble in the same manner as before:

```
>>> ensemble.fit(X[:75], y[:75])
Processing layers (3)

Fitting layer-1
[Parallel(n_jobs=-1)]: Done   7 out of   6 | elapsed:    0.1s remaining:   -0.0s
[Parallel(n_jobs=-1)]: Done   7 out of   6 | elapsed:    0.1s remaining:   -0.0s
[Parallel(n_jobs=-1)]: Done   7 out of   6 | elapsed:    0.1s remaining:   -0.0s
[Parallel(n_jobs=-1)]: Done   7 out of   6 | elapsed:    0.1s remaining:   -0.0s
[Parallel(n_jobs=-1)]: Done   7 out of   6 | elapsed:    0.1s remaining:   -0.0s
[Parallel(n_jobs=-1)]: Done   6 out of   6 | elapsed:    0.1s finished
layer-1 Done | 00:00:00

Fitting layer-2
[Parallel(n_jobs=-1)]: Done   7 out of   6 | elapsed:    0.0s remaining:   -0.0s
[Parallel(n_jobs=-1)]: Done   7 out of   6 | elapsed:    0.1s remaining:   -0.0s
[Parallel(n_jobs=-1)]: Done   7 out of   6 | elapsed:    0.1s remaining:   -0.0s
[Parallel(n_jobs=-1)]: Done   7 out of   6 | elapsed:    0.1s remaining:   -0.0s
[Parallel(n_jobs=-1)]: Done   7 out of   6 | elapsed:    0.1s remaining:   -0.0s
```

```
[Parallel(n_jobs=-1)]: Done   6 out of   6 | elapsed:    0.1s finished
layer-2 Done | 00:00:00

Fitting layer-3
[Parallel(n_jobs=-1)]: Done   1 out of   1 | elapsed:    0.0s finished
[Parallel(n_jobs=-1)]: Done   1 out of   1 | elapsed:    0.0s finished
layer-3 Done | 00:00:00

Fit complete | 00:00:00
```

Similarly with predictions:

```
>>> preds = ensemble.predict(X[75:])
Processing layers (3)

Predicting layer-1
[Parallel(n_jobs=-1)]: Done   2 out of   2 | elapsed:    0.0s finished
layer-1 Done | 00:00:00

Predicting layer-2
[Parallel(n_jobs=-1)]: Done   2 out of   2 | elapsed:    0.0s finished
layer-2 Done | 00:00:00

Predicting layer-3
[Parallel(n_jobs=-1)]: Done   1 out of   1 | elapsed:    0.0s finished
layer-3 Done | 00:00:00

Done | 00:00:00
```

The design of the `scores_` attribute allows an intuitive overview of how the base learner's perform in each layer.

```
>>> DataFrame(ensemble.scores_)
                                score_mean  score_std
layer-1 logisticregression        0.735420   0.156472
        randomforestclassifier    0.839260   0.055477
layer-2 logisticregression        0.668208   0.115576
        svc                       0.893314   0.001422
```

Model selection guide

The work horse class is the *Evaluator*, which allows you to grid search several models in one go across several preprocessing pipelines. The evaluator class pre-fits transformers, thus avoiding fitting the same preprocessing pipelines on the same data repeatedly.

The following example evaluates a Naive Bayes estimator and a K-Nearest-Neighbor estimator under three different preprocessing scenarios: no preprocessing, standard scaling, and subset selection. In the latter case, preprocessing is constituted by selecting a subset of features.

The scoring function

An important note is that the scoring function must be wrapped by *make_scorer()*, to ensure all scoring functions behave similarly regardless of whether they measure accuracy or errors. To wrap a function, simple do:

```
from mlens.metrics import make_scorer
f1_scorer = make_scorer(f1_score, average='micro', greater_is_better=True)
```

The `make_scorer` wrapper is a copy of the Scikit-learn's `sklearn.metrics.make_scorer()`, and you can import the Scikit-learn version as well. Note however that to pickle the *`Evaluator`*, you **must** import `make_scorer` from `mlens`.

A simple evaluation

Before throwing preprocessing into the mix, let's see how to evaluate a set of estimator. First, we need a list of estimator and a dictionary of parameter distributions that maps to each estimator. The estimators should be put in a list, either as is or as a named tuple (`(name, est)`). If you don't name the estimator, the *`Evaluator`* will automatically name the model as the class name in lower case. This name must be the key in the parameter dictionary. Let's see how to set this up:

```
from mlens.model_selection import Evaluator
from sklearn.naive_bayes import GaussianNB
from sklearn.neighbors import KNeighborsClassifier

from scipy.stats import randint

# Here we name the estimators ourselves
ests = [('gnb', GaussianNB()), ('knn', KNeighborsClassifier())]

# Now we map parameters to these
# The gnb doesn't have any parameters so we can skip it
pars = {'n_neighbors': randint(2, 20)}
params = {'knn': pars}
```

We can now run an evaluation over these estimators and parameter distributions by calling the `evaluate` method.

```
>>> evaluator = Evaluator(f1_scorer, cv=10, random_state=seed, verbose=1)
>>> evaluator.evaluate(X, y, ests, params, n_iter=10)
Evaluating 2 models for 10 parameter draws over 10 CV folds, totalling 200 fits
[Parallel(n_jobs=-1)]: Done 110 out of 110 | elapsed:    0.2s finished
Evaluation done | 00:00:00
```

The full history of the evaluation can be found in `cv_results`. To compare models with their best parameters, we can pass the `summary` attribute to a `pandas.DataFrame`.

```
>>> DataFrame(evaluator.summary)
        test_score_mean  test_score_std  train_score_mean  train_score_std  fit_time_
↪mean  fit_time_std              params
gnb          0.960000        0.032660          0.957037         0.005543          0.
↪001298      0.001131                  {}
knn          0.966667        0.033333          0.980000         0.004743          0.
↪000866      0.001001  {'n_neighbors': 15}
```

Preprocessing

Next, suppose we want to compare the models across a set of preprocessing pipelines. To do this, we first need to specify a dictionary of preprocessing pipelines to run through. Each entry in the dictionary should be a list of transformers to apply sequentially.

```
from mlens.preprocessing import Subset
from sklearn.preprocessing import StandardScaler
```

```
# Map preprocessing cases through a dictionary
preprocess_cases = {'none': [],
                    'sc': [StandardScaler()],
                    'sub': [Subset([0, 1])]
                    }
```

We can either fit the preprocessing pipelines and estimators in one go using the `fit` method, or we can pre-fit the transformers before we decide on estimators.

This can be helpful if the preprocessing is time-consuming, for instance if the preprocessing pipeline is an *EnsembleTransformer*. This class mimics how an ensemble creates prediction matrices during fit and predict calls, and can thus be used as a preprocessing pipeline to evaluate different candidate meta learners. See the *Ensemble model selection* tutorial for an example. To explicitly fit preprocessing pipelines, call `preprocess`.

```
>>> evaluator.preprocess(X, y, preprocess_cases)
Preprocessing 3 preprocessing pipelines over 10 CV folds
[Parallel(n_jobs=-1)]: Done  30 out of  30 | elapsed:    0.2s finished
Preprocessing done | 00:00:00
```

Model Selection across preprocessing pipelines

To evaluate the same set of estimators across all pipelines with the same parameter distributions, there is no need to take any heed of the preprocessing pipeline, just carry on as in the simple case:

```
>>> evaluator.evaluate(X, y, ests, params, n_iter=10)
>>> DataFrame(evaluator.summary)
          test_score_mean  test_score_std  train_score_mean  train_score_std  fit_
↪time_mean  fit_time_std               params
none gnb         0.960000        0.032660          0.957037         0.005543       0.
↪003507      0.003547                   {}
     knn         0.960000        0.044222          0.974815         0.007554       0.
↪002421      0.003270  {'n_neighbors': 11}
sc   gnb         0.960000        0.032660          0.957037         0.005543       0.
↪000946      0.000161                   {}
     knn         0.960000        0.044222          0.965185         0.003395       0.
↪000890      0.000568   {'n_neighbors': 8}
sub  gnb         0.780000        0.133500          0.791111         0.019821       0.
↪000658      0.000109                   {}
     knn         0.786667        0.122202          0.825926         0.016646       0.
↪000385      0.000063  {'n_neighbors': 11}
```

You can also map different estimators to different preprocessing folds, and map different parameter distribution to each case.

```
# We will map two different parameter distributions
pars_1 = {'n_neighbors': randint(20, 30)}
pars_2 = {'n_neighbors': randint(2, 10)}
params = {('sc', 'knn'): pars_1,
          ('none', 'knn'): pars_2,
          ('sub', 'knn'): pars_2}

# We can map different estimators to different cases
ests_1 = [('gnb', GaussianNB()), ('knn', KNeighborsClassifier())]
ests_2 = [('knn', KNeighborsClassifier())]
estimators = {'sc': ests_1,
```

```
              'none': ests_2,
              'sub': ests_1}
```

To run cross-validation, call the `evaluate` method. Make sure to specify the number of parameter draws to evaluate (the `n_iter` parameter).

```
>>> evaluator.evaluate(X, y, estimators, params, n_iter=10)
Evaluating 6 estimators for 10 parameter draws 10 CV folds, totalling 600 fits
[Parallel(n_jobs=-1)]: Done 600 out of 600 | elapsed:    1.0s finished
Evaluation done | 00:00:01
```

As before, we can summarize the evaluation in a nice `DataFrame`.

```
>>> DataFrame(evaluator.summary)
       test_score_mean  test_score_std  train_score_mean  train_score_std    fit_time_
↪mean  fit_time_std               params
none knn         0.966667        0.044721          0.960741         0.007444
↪0.001718      0.003330   {'n_neighbors': 3}
sc   gnb         0.960000        0.032660          0.957037         0.005543
↪0.000926      0.000139                 {}
     knn         0.940000        0.055377          0.962963         0.005738
↪0.000430      0.000035  {'n_neighbors': 20}
sub  gnb         0.780000        0.133500          0.791111         0.019821
↪0.000869      0.000126                 {}
     knn         0.800000        0.126491          0.837037         0.014815
↪0.000426      0.000068   {'n_neighbors': 9}
```

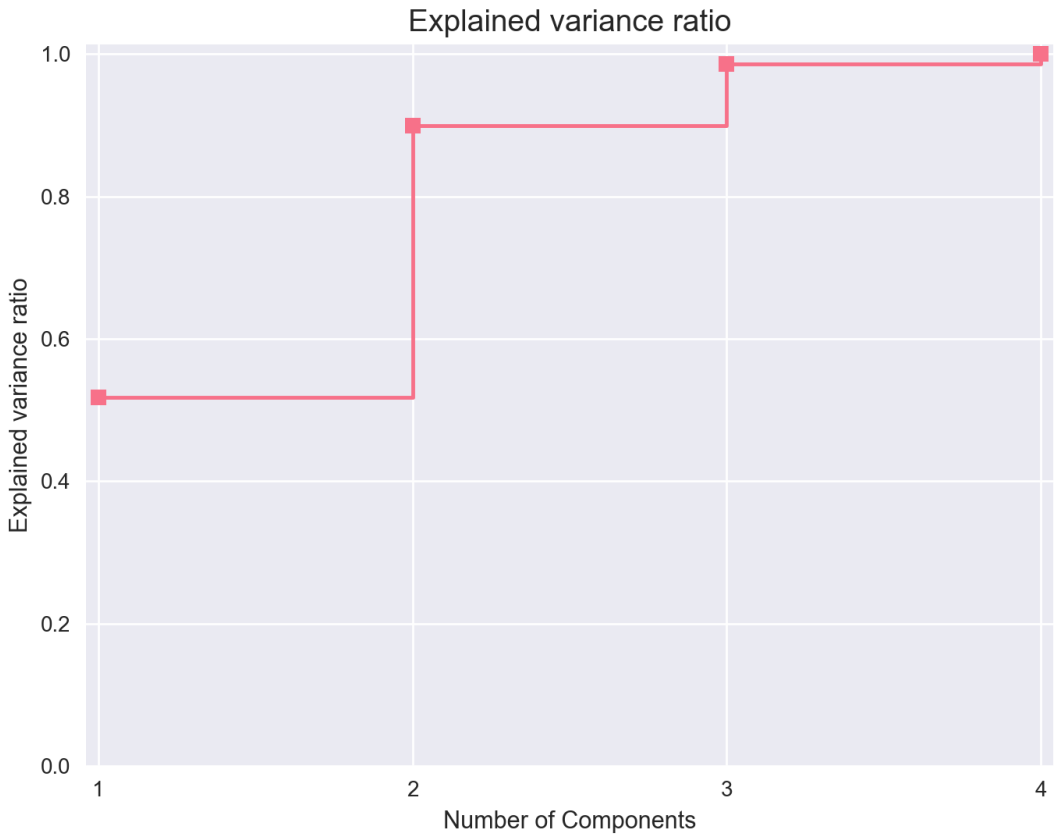
The `Evaluator` provides a one-stop-shop for comparing many different models in various configurations, and is a critical tool to leverage when building complex ensembles. It is especially helpful in combination with the *`EnsembleTransformer`*, which allows use to evaluate the next layer of an ensemble or a set of potential meta learners without having to run the entire ensemble every time. As such, it provides a way to perform greedy layer-wise parameter tuning. For more details, see the *Ensemble model selection* tutorial.

Visualization guide

Explained variance plot

The *`exp_var_plot`* function plots the explained variance from mapping a matrix `X` onto a smaller dimension using a user-supplied transformer, such as the Scikit-learn `sklearn.decomposition.PCA` transformer for Principal Components Analysis.

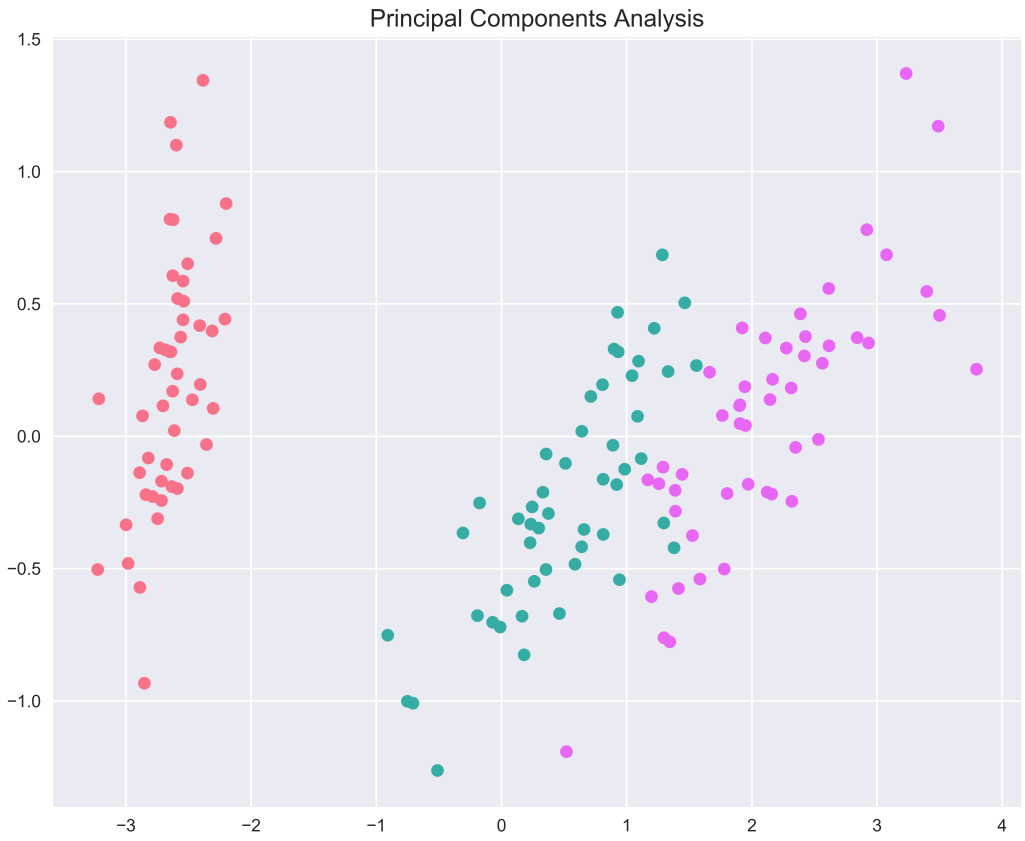
```
>>> from mlens.visualization import exp_var_plot
>>> from sklearn.decomposition import PCA
>>>
>>> exp_var_plot(X, PCA(), marker='s', where='post')
```



Principal Components Analysis plot

The *pca_plot* function plots a PCA analysis or similar if `n_components` is one of `[1, 2, 3]`. By passing a class labels, the plot shows how well separated different classes are.

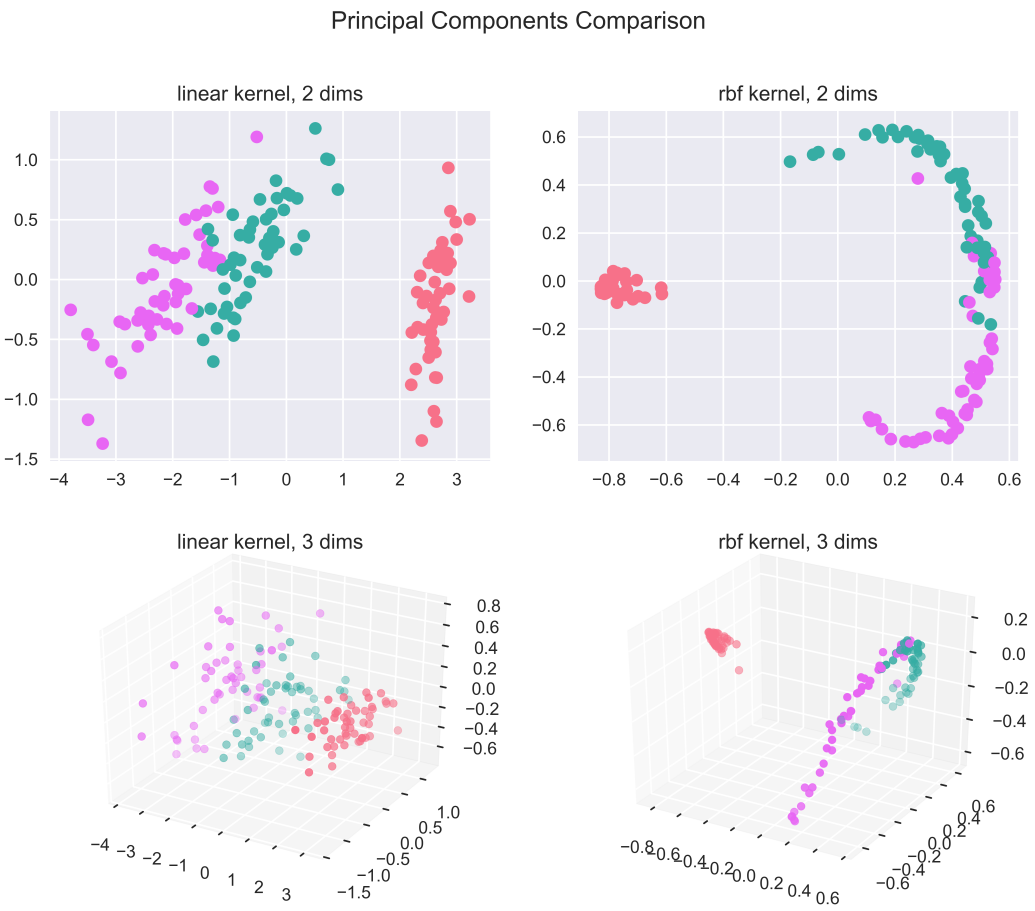
```
>>> from mlens.visualization import pca_plot
>>> from sklearn.decomposition import PCA
>>>
>>> pca_plot(X, PCA(n_components=2), y=y)
```



Principal Components Comparison plot

The *pca_comp_plot* function plots a matrix of PCA analyses, one for each combination of `n_components=2, 3` and `kernel='linear', 'rbf'`.

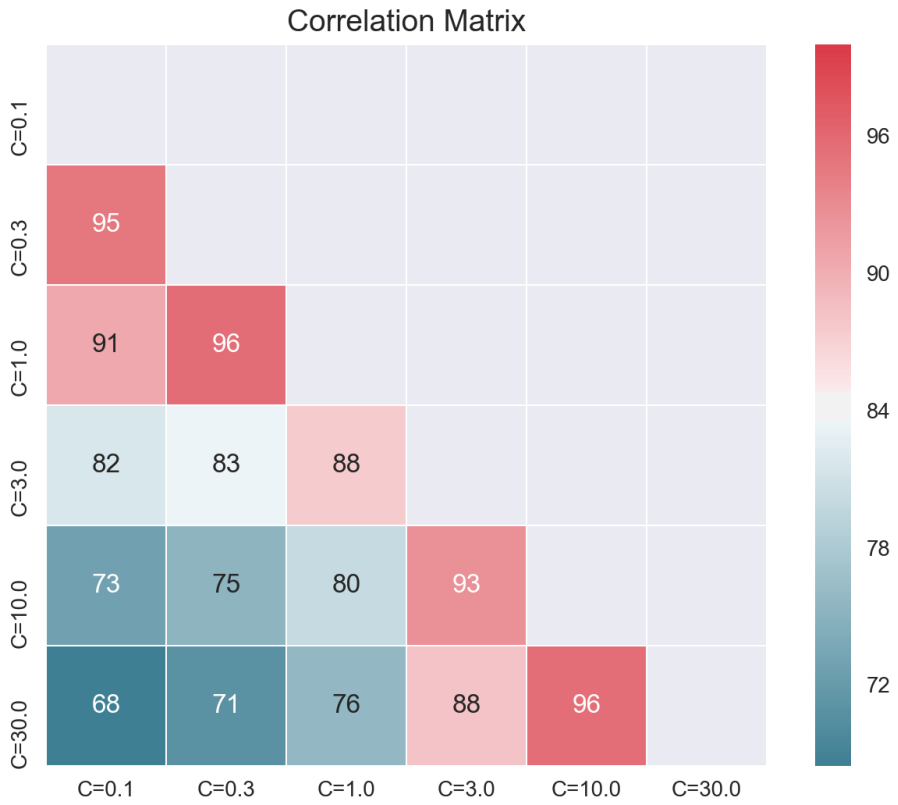
```
>>> from mlens.visualization import pca_comp_plot
>>>
>>> pca_comp_plot (X, y)
```



Correlation matrix plot

The *corrmat* function plots the lower triangle of a correlation matrix and is adapted the Seaborn correlation matrix.

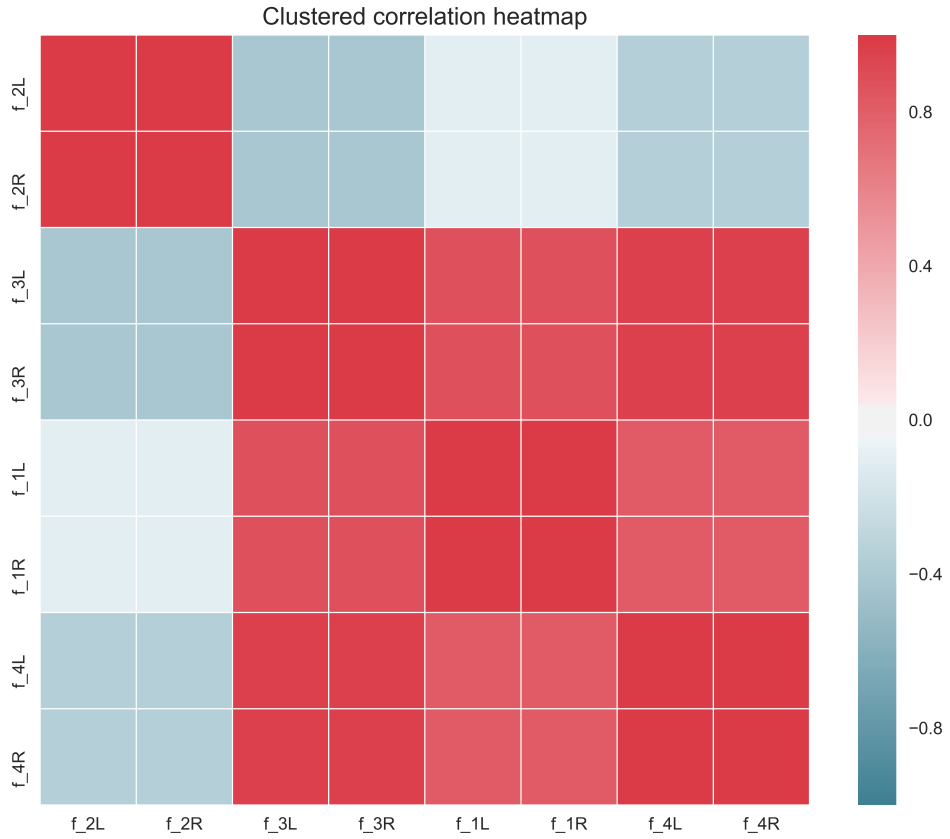
```
>>> from mlens.visualization import corrmat
>>>
>>> # Generate som different predictions to correlate
>>> params = [0.1, 0.3, 1.0, 3.0, 10, 30]
>>> preds = np.zeros((150, 6))
>>> for i, c in enumerate(params):
...     preds[:, i] = LogisticRegression(C=c).fit(X, y).predict(X)
>>>
>>> corr = DataFrame(preds, columns=['C=%.1f' % i for i in params]).corr()
>>> corrmat(corr)
```



Clustered correlation heatmap plot

The *clustered_corrmap* function is similar to *corrmat*, but differs in two respects. First, and most importantly, it uses a user supplied clustering estimator to cluster the correlation matrix on similar features, which can often help visualize whether there are blocks of highly correlated features. Secondly, it plots the full matrix (as opposed to the lower triangle).

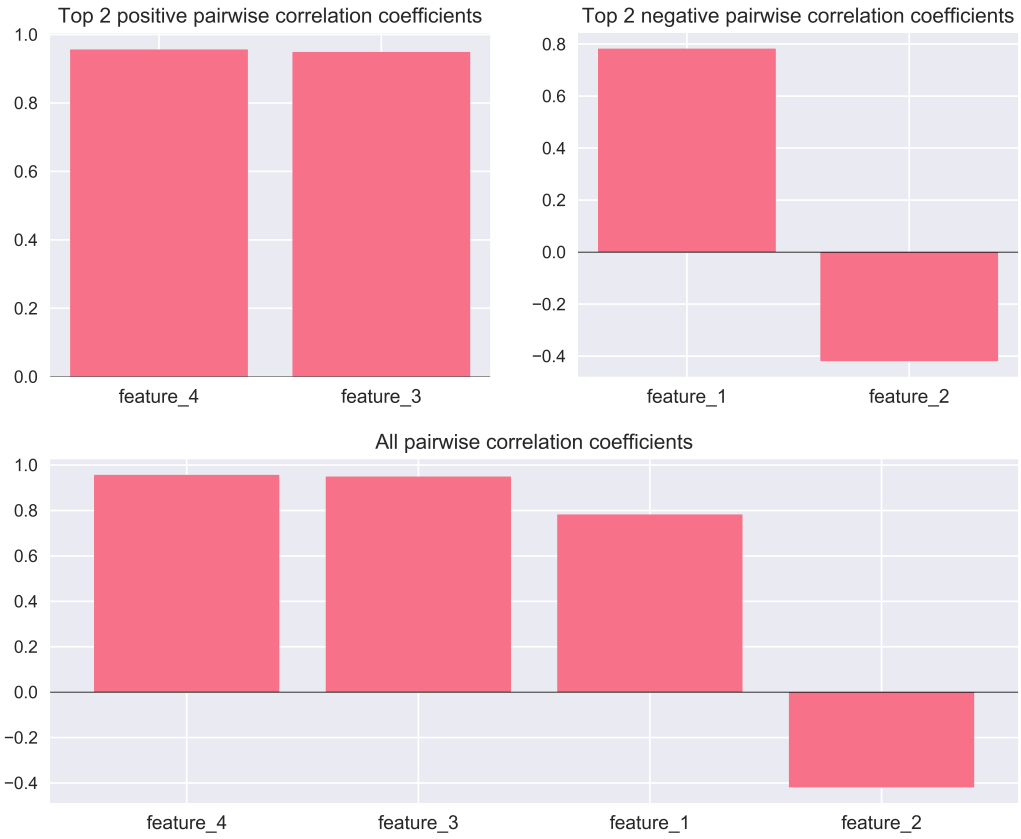
```
>>> from mlens.visualization import clustered_corrmap
>>> from sklearn.cluster import KMeans
>>>
>>> Z = DataFrame(X, columns=['f_%i' %i for i in range(1, 5)])
>>>
>>> # We duplicate all features, note that the heatmap orders features
>>> # as duplicate pairs, and thus fully pick up on this duplication.
>>> corr = Z.join(Z, lsuffix='L', rsuffix='R').corr()
>>>
>>> clustered_corrmap(corr, KMeans())
```



Input-Output correlations

The *corr_X_y* function gives a dashboard of pairwise correlations between the input data (`X`) and the labels to be predicted (`y`). If the number of features is large, it is advised to set the `no_ticks` parameter to `True`, to avoid rendering an illegible x-axis. Note that `X` must be a `pandas.DataFrame`.

```
>>> Z = DataFrame(X, columns=['feature_%i' %i for i in range(1, 5)])
>>> corr_X_y(Z, y, 2, no_ticks=False)
```



1.5.4 Tutorials

The following tutorials highlight advanced functionality and provide in-depth material on ensemble APIs.

Tutorial	Content
Propagating input features	Propagate feature input features through layers
	to allow several layers to see the same input.
Probabilistic ensemble learning	Build layers that output class probabilities from each base
	learner so that the next layer or meta estimator learns
	from probability distributions.
Advanced Subsemble techniques	Learn homogenous partitions of feature space
	that maximize base learner's performance on each partition.
General multi-layer ensemble learning	How to build ensembles with different layer classes
Passing file paths as data input	Avoid loading data into the parent process by specifying a
	file path to a memmaped array or a csv file.
Ensemble model selection	Build transformers that replicate layers in ensembles for
	model selection of higher-order layers and / or meta learners.

We use the same preliminary settings as in the *getting started* section.

Propagating input features

When stacking several layers of base learners, the variance of the input will typically get smaller as learners get better and better at predicting the output and the remaining errors become increasingly difficult to correct for. This multicolinearity can significantly limit the ability of the ensemble to improve upon the best score of the subsequent layer as there is too little variation in predictions for the ensemble to learn useful combinations. One way to increase this variation is to propagate features from the original input and / or earlier layers. To achieve this in ML-Ensemble, we use the `propagate_features` attribute. To see how this works, let's compare a three-layer ensemble with and without feature propagation.

```
from mlens.ensemble import SuperLearner
from sklearn.linear_model import LogisticRegression
from sklearn.ensemble import RandomForestClassifier
from sklearn.svm import SVC

def build_ensemble(incl_meta, propagate_features=None):
    """Return an ensemble."""
    if propagate_features:
        n = len(propagate_features)
        propagate_features_1 = propagate_features
        propagate_features_2 = [i for i in range(n)]
    else:
        propagate_features_1 = propagate_features_2 = None

    estimators = [RandomForestClassifier(random_state=seed), SVC()]

    ensemble = SuperLearner()
    ensemble.add(estimators, propagate_features=propagate_features_1)
    ensemble.add(estimators, propagate_features=propagate_features_2)

    if incl_meta:
        ensemble.add_meta(LogisticRegression())
    return ensemble
```

Without feature propagation, the meta learner will learn from the predictions of the penultimate layers:

```
>>> base = build_ensemble(False)
>>> base.fit(X, y)
>>> base.predict(X)
array([[ 2.,  2.],
       [ 2.,  2.],
       [ 2.,  2.],
       [ 1.,  1.],
       [ 1.,  1.]])
```

When we propagate features, some (or all) of the input seen by one layer is passed along to the next layer. For instance, we can propagate some or all of the input array through our two intermediate layers to the meta learner input of the meta learner:

```
>>> base = build_ensemble(False, [1, 3])
>>> base.fit(X, y)
>>> base.predict(X)
array([[ 3.20000005,  2.29999995,  2.        ,  2.        ],
       [ 3.20000005,  2.29999995,  2.        ,  2.        ],
       [ 3.        ,  2.0999999 ,  2.        ,  2.        ],
       [ 3.20000005,  1.5       ,  1.        ,  1.        ],
       [ 2.79999995,  1.39999998,  1.        ,  1.        ]])
```

In this scenario, the meta learner will see noth the predictions made by the penultimate layer, as well as the second and fourth feature of the original input. By propagating features, the issue of multicolinearity in deep ensembles can be mitigated. In particular, it can give the meta learner greater opportunity to identify neighborhoods in the original feature space where base learners struggle. We can get an idea of how feature propagation works with our toy example. First, we need a simple ensemble evaluation routine.

```
def evaluate_ensemble(propagate_features):
    """Wrapper for ensemble evaluation."""
    ens = build_ensemble(True, propagate_features)
    ens.fit(X[:75], y[:75])
    pred = ens.predict(X[75:])
    return f1_score(pred, y[75:], average='micro')
```

In our case, propagating the original features through two layers of the same library of base learners gives a dramatic increase in performance on the test set:

```
>>> score_no_prep = evaluate_ensemble(None)
>>> score_prep = evaluate_ensemble([0, 1, 2, 3])
>>> print("Test set score no feature propagation  : %.3f" % score_no_prep)
>>> print("Test set score with feature propagation: %.3f" % score_prep)
Test set score no feature propagation  : 0.666
Test set score with feature propagation: 0.987
```

By combining feature propagation with the *Subset* transformer, you can propagate the feature through several layers without any of the base estimators in those layers seeing the propagated features. This can be desirable if you want to propagate the input features to the meta learner without intermediate base learners always having access to the original input data. In this case, we specify propagation as above, but add a preprocessing pipeline to intermediate layers:

```
from mlens.preprocessing import Subset

estimators = [RandomForestClassifier(random_state=seed), SVC()]
ensemble = SuperLearner()

# Initial layer, propagate as before
ensemble.add(estimators, propagate_features=[0, 1])

# Intermediate layer, keep propagating, but add a preprocessing
# pipeline that selects a subset of the input
ensemble.add(estimators,
             preprocessing=[Subset([2, 3])],
             propagate_features=[0, 1])
```

In the above example, the two first features of the original input data will be propagated through both layers, but the second layer will not be trained on it. Instead, it will only see the predictions made by the base learners in the first layer.

```
>>> ensemble.fit(X, y)
>>> n = ensemble.layer_2.estimators_[0][1][1].feature_importances_.shape[0]
>>> m = ensemble.predict(X).shape[1]
>>> print("Num features seen by estimators in intermediate layer: %i" % n)
>>> print("Num features in the output array of the intermediate layer: %i" % m)
Num features seen by estimators in intermediate layer: 2
Num features in the output array of the intermediate layer: 4
```

Probabilistic ensemble learning

When the target to predict is a class label, it can often be beneficial to let higher-order layers or the meta learner learn from *class probabilities*, as opposed to the predicted class. Scikit-learn classifiers can return a matrix that, for each observation in the test set, gives the probability that the observation belongs to the a given class. While we are ultimately interested in class membership, this information is much richer that just feeding the predicted class to the meta learner. In essence, using class probabilities allow the meta learner to weigh in not just the predicted class label (the highest probability), but also with what confidence each estimator makes the prediction, and how estimators consider the alternative. First, let us set a benchmark ensemble performance when learning is by predicted class membership.

```
from mlens.ensemble import BlendEnsemble
from sklearn.linear_model import LogisticRegression
from sklearn.ensemble import RandomForestClassifier
from sklearn.svm import SVC

def build_ensemble(proba, **kwargs):
    """Return an ensemble."""
    estimators = [RandomForestClassifier(random_state=seed),
                  SVC(probability=proba)]

    ensemble = BlendEnsemble(**kwargs)
    ensemble.add(estimators, proba=proba)    # Specify 'proba' here
    ensemble.add_meta(LogisticRegression())

    return ensemble
```

As in the *ensemble guide*, we fit on the first half, and test on the remainder.

```
>>> ensemble = build_ensemble(proba=False)
>>> ensemble.fit(X[:75], y[:75])
>>> preds = ensemble.predict(X[75:])
>>> f1_score(preds, y[75:], average='micro')
0.69333333333333336
```

Now, to enable probabilistic learning, we set `proba=True` in the `add` method for all layers except the final meta learner layer.

```
>>> ensemble = build_ensemble(proba=True)
>>> ensemble.fit(X[:75], y[:75])
>>> preds = ensemble.predict(X[75:])
>>> print('Prediction shape: {}'.format(preds.shape))
>>> f1_score(preds, y[75:], average='micro')
Prediction shape: (75,)
0.97333333333333338
```

In this case, using probabilities has a drastic effect on predictive performance, increasing some 40 percentage points. As a final remark, if you want the *ensemble* to return predicted probabilities, specify the final layer using the `add` method with `meta=True`.

Advanced Subsemble techniques

Subsembles leverages the idea that neighborhoods of feature space have a specific local structure. When we fit an estimator across all feature space, it is very hard to capture several such local properties. Subsembles partition the feature space and fits each base learner to each partitions, thereby allow base learners to optimize locally. Instead, the task of generalizing across neighborhoods is left to the meta learner. This strategy can be very powerful when the

local structure first needs to be extracted, before an estimator can learn to generalize. Suppose you want to learn the probability distribution of some variable y. Often, the true distribution is multi-modal, which is an extremely hard problem. In fact, most machine learning algorithms, especially with convex optimization objectives, are ill equipped to solve this problem. Subsembles can overcome this issue allowing base estimators to fit one mode of the distribution at a time, which yields a better representation of the distribution and greatly facilitates the learning problem of the meta learner.

By default, the *Subsemble* class partitioning the dataset randomly. Note however that partitions are created on the data "as is", so if the ordering of observations is not randomly, neither will the partitioning be. For this reason, it is recommended to shuffle the data (e.g. via the `shuffle` option at instantiation). To build a subsemble with random partitions, the only parameter to consider is the number of `partitions` when instantiating the *Subsemble*.

```
from mlens.ensemble import Subsemble
from sklearn.linear_model import LogisticRegression
from sklearn.svm import SVC

def build_subsemble():
    """Build a subsemble with random partitions"""
    sub = Subsemble(partitions=3, folds=2)
    sub.add([SVC(), LogisticRegression()])
    return sub
```

During training, the base learners are copied to each partition, so the output of each layer gets multiplied by the number of partitions. In this case, we have 2 base learners for 3 partitions, giving 6 prediction features.

```
>>> sub = build_subsemble()
>>> sub.fit(X, y)
>>> sub.predict(X[:10])shape
(10, 6)
```

By creating partitions, subsembles scale significantly better than the *SuperLearner*, but in contrast to *BlendEnsemble*, the full training data is leveraged during training. But randomly partitioning the data does however not exploit the full advantage of locality, since it is only by luck that we happen to create such partitions. A better way is to *learn* how to best partition the data. We can either use unsupervised algorithms to generate clusters, or supervised estimators and create partitions based on their predictions. In ML-Ensemble, this is achieved by passing an estimator as `partition_estimator`. This estimator can differ between layers.

Very few limitation are imposed on the estimator: it must have a `fit` method that takes `X` (and possibly `y`) as inputs, and there must be a method that generates class labels (i.e. partition ids) to a passed dataset. The default method is `predict`, but you can specify another method with the `attr` option when adding a layer, and which data to use with this method (`partition_on='X', 'y', 'both'`). This level of generality does impose some responsibility on the user. In particular, it is up to the user to ensure that sensible partitions are created. Problems to watch out for is too small partitions (too many clusters, too uneven cluster sizes) and clusters with too little variation: for instance with only a single class label in the entire partition, base learners have nothing to learn.

Let's see how to do this in practice. For instance, we can use an unsupervised K-Means clustering estimator to partition the data, like so:

```
from sklearn.cluster import KMeans

def build_clustered_subsemble(estimator):
    """Build a subsemble with random partitions"""
    sub = Subsemble(partitions=2,
                    partition_estimator=estimator,
                    folds=2)

    sub.add([SVC(), LogisticRegression()])
```

```
    sub.add_meta(SVC())
    return sub
```

Note that the `sklearn.cluster.KMeans` estimator generates class labels through the `predict` method. To build a subsemble with K-Means clustering we carry on as usual:

```
>>> sub = build_clustered_subsemble(KMeans(2))
>>> sub.fit(X[:, [0, 1]], y)
```

In our toy example, fitting the KMeans estimator on all data leads to completely separated class clusters, so each partition has not output variation. For this reason, we had to fit on only the two first columns. But this is not a very good way of doing it: instead, we should customize the partitioning estimator. For instance, we can use Scikit-learn's `sklearn.pipeline.Pipeline` class to put a dimensionality reduction transformer before the partitioning estimator, such as a `sklearn.decomposition.PCA`, or the *`mlens.preprocessing.Subset`* transformer to drop some features before estimation.

```
from mlens.preprocessing import Subset
from sklearn.pipeline import make_pipeline

pe = make_pipeline(Subset([0, 1]), KMeans(2))
sub = build_clustered_subsemble(pe)
```

This subsemble can now be fitted on all data: the clustering algorithm will only see the first two features, but the base learners will be trained on all data.

```
>>> sub.fit(X, y)
```

In general, you may need to wrap an estimator around a custom class to modify it's output to generate good partitions. For instance, in regression problems, the output of a supervised estimator needs to be binarized to give a discrete number of partitions. Here's minimalist way of wrapping a Scikit-learn estimator:

```
from sklearn.linear_model import LinearRegression

class MyClass(LinearRegression):

    def __init__(self, **kwargs):
        super(MyClass, self).__init__(**kwargs)

    def fit(self, X, y):
        """Fit estimator."""
        super(MyClass, self).fit(X, y)
        return self

    def predict(self, X):
        """Generate partition"""
        p = super(MyClass, self).predict(X)
        return 1 * (p > p.mean())
```

By default, the Subsemble will call the `fit` method of the partition estimator separately first, then the `predict` (or otherwise specified) method. To avoid calling `fit`, pass `fit_estimator=False` when adding the layer. Finally, to summarize the functionality in one example, let's implement a simple (but rather useless) partition estimator that splits the data in half based on the sum of the features.

```
class SimplePartitioner():

    def __init__(self):
        pass
```

```
    def our_custom_function(self, X, y=None):  # strictly, speaking, y can be omitted
        """Split the data in half based on the sum of features"""
        # Labels should be numerical
        return 1 * (X.sum(axis=1) > X.sum(axis=1).mean())
```

To build the ensemble, we need specify that we don't want to fit the estimator, and that `our_custom_function` should be called for partitioning. An important note is that the number of partitions the estimator creates *must* match the `partitions` argument of the Subsemble. In contrast, the `folds` option is completely independent.

```
>>> sub = Subsemble(partitions=2, folds=3)
>>> sub.add([SVC(), LogisticRegression()],
...         partition_estimator=SimplePartitioner(),
...         fit_estimator=False,
...         attr="our_custom_function")
>>> sub.fit(X, y)
```

A final word of caution. When implementing custom estimators from scratch, some care needs to be taken if you plan on copying the Subsemble. It is advised that the estimator inherits the `sklearn.base.BaseEstimator` class to provide a Scikit-learn compatible interface. For further information, see the *API* documentation of the *Subsemble* and *mlens.base.indexer.ClusteredSubsetIndex*.

General multi-layer ensemble learning

The modular `add` API of ML-Ensembles allow users to build arbitrarily deep ensembles. If you would like to alternate between the *type* of each layer the *SequentialEnsemble* class can be used to specify what type of layer (i.e. stacked, blended, subsamle-style) to add. This can be particularly powerful if facing a large dataset, as the first layer can use a fast appraoch such as blending, while subsequent layers fitted on the remaining data can use more computationally intensive approaches. The type of layer, along with any parameter settings pertaining to that layer, are specified in the `add` method.

```
from mlens.ensemble import SequentialEnsemble

ensemble = SequentialEnsemble()

# The initial layer is a the same as a BlendEnsemble with one layer
ensemble.add('blend', [SVC(), RandomForestClassifier(random_state=seed)])

# The second layer is a the same as a SuperLearner with one layer
ensemble.add('stack', [SVC(), RandomForestClassifier(random_state=seed)])

# The meta estimator is added as in any other ensemble
ensemble.add_meta(SVC())
```

Note that currently, the sequential ensemble uses the backend terminology and may not overlap with what the ensemble classes uses. This will be fixed in a coming release. Until then, the following conversion may be helpful.

front-end parameter	SequentialEnsemble parameter
'SuperLearner'	'stack'
'BlendEnsemble'	'blend'
'Subsemble'	'subset'
'folds'	'n_splits'
'partitions'	'n_partitions'

This ensemble can now be used for fitting and prediction with the conventional syntax.

```
>>> preds = ensemble.fit(X[:75], y[:75]).predict(X[75:])
>>> f1_score(preds, y[75:], average='micro')
0.97333333333333338
```

In this case, the multi-layer *SequentialEnsemble* with an initial blended layer and second stacked layer achieves similar performance as the *BlendEnsemble* with probabilistic learning. Note that we could have made any of the layers probabilistic by setting `Proba=True`.

Passing file paths as data input

With large datasets, it can be expensive to load the full data into memory as a numpy array. Since ML-Ensemle uses a memmaped cache, the need to keep the full array in memory can be entirely circumvented by passing a file path as entry to `X` and `y`. There are two important things to note when doing this.

First, ML-Ensemble delpoys Scikit-learn's array checks, and passing a string will cause an error. To avoid this, the ensemble must be initialized with `array_check=0`, in which case there will be no checks on the array. The user should make certain that the the data is approprate for esitmation, by converting missing values and infinites to numerical representation, ensuring that all features are numerical, and remove any headers, index columns and footers.

Second, ML-Ensemble expects the file to be either a `csv`, an `npy` or `mmap` file and will treat these differently.

- If a path to a `csv` file is passed, the ensemble will first **load** the file into memory, then dump it into the cache, before discarding the file from memory by replacing it with a pointer to the memmaped file. The loading module used for the `csv` file is the `numpy.loadtxt()` function.
- If a path to a `npy` file is passed, a memmaped pointer to it will be loaded.
- If a path to a `mmap` file is passed, it will be used as the memmaped input array for estimation.

```
import os
import gc
import tempfile

# We create a temporary folder in the current working directory
temp = tempfile.TemporaryDirectory(dir=os.getcwd())

# Dump the X and y array in the temporary directory, here as csv files
fx = os.path.join(temp.name, 'X.csv')
fy = os.path.join(temp.name, 'y.csv')

np.savetxt(fx, X)
np.savetxt(fy, y)
```

We can now fit any ensemble simply by passing the file pointers `fx` and `fy`. Remember to set `array_check=0`.

```
>>> ensemble = build_ensemble(False, array_check=0)
>>> ensemble.fit(fx, fy)
>>> preds = ensemble.predict(fx)
>>> preds[:10]
array([ 2.,  2.,  2.,  1.,  1.,  2.,  2.,  2.,  2.,  2.])
```

If you are following the examples on your machine, don't forget to remove the temporary directory.

```
try:
    temp.cleanup()
    del temp
except OSError:
```

```
    # This can fail on Windows
    pass
```

Ensemble model selection

Ensembles benefit from a diversity of base learners, but often it is not clear how to parametrize the base learners. In fact, combining base learners with lower predictive power can often yield a superior ensemble. This hinges on the errors made by the base learners being relatively uncorrelated, thus allowing a meta estimator to learn how to overcome each model's weakness. But with highly correlated errors, there is little for the ensemble to learn from.

To fully exploit the learning capacity in an ensemble, it is beneficial to conduct careful hyper parameter tuning, treating the base learner's parameters as the parameters of the ensemble. By far the most critical part of the ensemble is the meta learner, but selecting an appropriate meta learner can be an ardous task if the entire ensemble has to be evaluated each time.

The *EnsembleTransformer* can be leveraged to treat the initial layers of the ensemble as preprocessing. Thus, a copy of the transformer is fitted once on each fold, and any model selection will use these pre-fits to convert raw input to prediction matrices that corresponds to the output of the specified ensemble.

The transformer follows the same API as the *SequentialEnsemble*, but does not implement a meta estimator and has a transform method that recovers the prediction matrix from the `fit` call. In the following example, we run model selection on the meta learner of a blend ensemble, and try two configurations of the blend ensemble: learning from class predictions or from probability distributions over classes.

```
from mlens.preprocessing import EnsembleTransformer
from mlens.model_selection import Evaluator
from scipy.stats import uniform, randint
from pandas import DataFrame

# Set up two competing ensemble bases as preprocessing transformers:
# one blend ensemble base with proba and one without
base_learners = [RandomForestClassifier(random_state=seed),
                 SVC(probability=True)]

proba_transformer = EnsembleTransformer().add('blend', base_learners, proba=True)
class_transformer = EnsembleTransformer().add('blend', base_learners, proba=False)

# Set up a preprocessing mapping
# Each pipeline in this map is fitted once on each fold before
# evaluating candidate meta learners.
preprocessing = {'proba': [('layer-1', proba_transformer)],
                 'class': [('layer-1', class_transformer)]}

# Set up candidate meta learners
# We can specify a dictionary if we wish to try different candidates on
# different cases, or a list if all estimators should be run on all
# preprocessing pipelines (as in this example)
meta_learners = [SVC(), ('rf', RandomForestClassifier(random_state=2017))]

# Set parameter mapping
# Here, we differentiate distributions between cases for the random forest
params = {'svc': {'C': uniform(0, 10)},
          ('class', 'rf'): {'max_depth': randint(2, 10)},
          ('proba', 'rf'): {'max_depth': randint(2, 10),
                            'max_features': uniform(0.5, 0.5)}
          }
```

```
evaluator = Evaluator(scorer=f1, random_state=2017, cv=20)

evaluator.fit(X, y, meta_learners, params, preprocessing=preprocessing, n_iter=2)
```

We can now compare the performance of the best fit for each candidate meta learner.

```
>>> DataFrame(evaluator.summary)
           test_score_mean  test_score_std  train_score_mean  train_score_std  fit_
↪time_mean  fit_time_std                                              params
class rf          0.955357        0.060950          0.972535         0.008303       0.
↪024585      0.014300                                   {'max_depth': 5}
      svc         0.961607        0.070818          0.972535         0.008303       0.
↪000800      0.000233                                  {'C': 7.67070164682}
proba rf          0.980357        0.046873          0.992254         0.007007       0.
↪022789      0.003296   {'max_depth': 3, 'max_features': 0.883535082341}
      svc         0.974107        0.051901          0.969718         0.008060       0.
↪000994      0.000367                                  {'C': 0.209602254061}
```

In this toy example, our model selection suggests the Random Forest is the best meta learner when the ensemble uses probabilistic learning.

1.5.5 Ensemble classes

ML-Ensemble implements four types of ensembles:

- *Super Learner* (stacking)
- *Subsemble*
- *Blend Ensemble*
- *Sequential Ensemble*

Each ensemble class can be built with several layers, and each layer can output class probabilities if desired. The `SequentialEnsemble` class is a generic ensemble class that allows the user to mix types between layers, for instance by setting the first layer to a Subsemble and the second layer to a Super Learner. Here, we will briefly introduce ensemble specific parameters and usage. For full documentation, see the *API* section.

Super Learner

The `SuperLearner` (also known as a Stacking Ensemble) is an supervised ensemble algorithm that uses K-fold estimation to map a training set (X, y) into a prediction set (Z, y), where the predictions in Z are constructed using K-Fold splits of X to ensure Z reflects test errors, and that applies a user-specified meta learner to predict y from Z.

The main parameter to specify is the `folds` parameter that determines the number of folds to use during cross-validation. The algorithm in sudo code follows:

1. Specify a library L of base learners
2. Fit all base learners on X and store the fitted estimators.
3. Split X into K folds, fit every learner in L on the training set and predict test set. Repeat until all folds have been predicted.
4. Construct a matrix Z by stacking the predictions per fold.
5. Fit the meta learner on Z and store the learner

The ensemble can be used for prediction by mapping a new test set T into a prediction set Z' using the learners fitted in (2), and then mapping Z' to y' using the fitted meta learner from (5).

The Super Learner does asymptotically as well as (up to a constant) an Oracle selector. For the theory behind the Super Learner, see[1] and[2] as well as references therein.

Stacking K-fold predictions to cover an entire training set is a time consuming method and can be prohibitively costly for large datasets. With large data, other ensembles that fits an ensemble on subsets can achieve similar performance at a fraction of the training time. However, when data is noisy or of high variance, the *SuperLearner* ensure all information is used during fitting.

References

Notes

This implementation uses the agnostic meta learner approach, where the user supplies the meta learner to be used. For the original Super Learner algorithm (i.e. learn the best linear combination of the base learners), the user can specify a linear regression as the meta learner.

Subsemble

Subsemble is a supervised ensemble algorithm that uses subsets of the full data to fit a layer, and within each subset K-fold estimation to map a training set (X, y) into a prediction set (Z, y), where Z is a matrix of prediction from each estimator on each subset (thus of shape `[n_samples, (n_partitions * n_estimators)]`). Z is constructed using K-Fold splits of each partition of X to ensure Z reflects test errors within each partition. A final user-specified meta learner is fitted to the final ensemble layer's prediction, to learn the best combination of subset-specific estimator predictions.

The main parameters to consider is the number of `partitions`, which will increase the number of estimators in the layer by a factor of the number of base learners specified, and the number of `folds` to be used during cross validation in each partition.

The algorithm in sudo code follows:

1. For each layer in the ensemble, do:
   (a) Specify a library of L base learners
   (b) Specify a partition strategy and partition X into J subsets.
   (c) For each partition do:
      i. Fit all base learners and store them
      ii. Create K folds
      iii. For each fold, do:
         A. Fit all base learners on the training folds
         B. Collect *all* test folds, across partitions, and predict.
   (d) Assemble a cross-validated prediction matrix $Z \in \mathbb{R}^{(n \times (L \times J))}$ by stacking predictions made in the cross-validation step.

[1] van der Laan, Mark J.; Polley, Eric C.; and Hubbard, Alan E., "Super Learner" (July 2007). U.C. Berkeley Division of Biostatistics Working Paper Series. Working Paper 222. http://biostats.bepress.com/ucbbiostat/paper222

[2] Polley, Eric C. and van der Laan, Mark J., "Super Learner In Prediction" (May 2010). U.C. Berkeley Division of Biostatistics Working Paper Series. Working Paper 266. http://biostats.bepress.com/ucbbiostat/paper266

2. Fit the meta learner on Z and store the learner.

The ensemble can be used for prediction by mapping a new test set T into a prediction set Z' using the learners fitted in (1.3.1), and then using Z' to generate final predictions through the fitted meta learner from (2).

The Subsemble does asymptotically as well as (up to a constant) the Oracle selector. For the theory behind the Subsemble, see[3] and references therein.

By partitioning the data into subset and fitting on those, a Subsemble can reduce training time considerably if estimators does not scale linearly. Moreover, Subsemble allows estimators to learn different patterns from each subset, and so can improve the overall performance by achieving a tighter fit on each subset. Since all observations in the training set are predicted, no information is lost between layers.

References

Notes

This implementation splits X into partitions sequentially, i.e. without randomizing indices. To achieve randomized partitioning, set `shuffle` to `True`. Supervised partitioning is under development.

Blend Ensemble

The *`BlendEnsemble`* is a supervised ensemble closely related to the *`SuperLearner`*. It differs in that to estimate the prediction matrix Z used by the meta learner, it uses a subset of the data to predict its complement, and the meta learner is fitted on those predictions.

The user must specify how much of the data should be used to train the layer, `test_size`, and how much should be held out for prediction. Prediction for the held-out set are passed to the next layer or meta estimator, so information is with each layer.

By only fitting every base learner once on a subset of the full training data, *`BlendEnsemble`* is a fast ensemble that can handle very large datasets simply by only using portion of it at each stage. The cost of this approach is that information is thrown out at each stage, as one layer will not see the training data used by the previous layer.

With large data that can be expected to satisfy an i.i.d. assumption, the *`BlendEnsemble`* can achieve similar performance to more sophisticated ensembles at a fraction of the training time. However, with data data is not uniformly distributed or exhibits high variance the *`BlendEnsemble`* can be a poor choice as information is lost at each stage of fitting.

Sequential Ensemble

The *`SequentialEnsemble`* allows users to build ensembles with different classes of layers. Instead of setting parameters upfront during instantiation, the user specified parameters for each layer when calling `add`. The user must thus specify what type of layer is being added (blend, super learner, subsemble), estimators, preprocessing if applicable, and any layer-specific parameters. The Sequential ensemble is best illustrated through an example:

```
>>> from mlens.ensemble import SequentialEnsemble
>>> from mlens.metrics.metrics import rmse
>>> from sklearn.datasets import load_boston
>>> from sklearn.linear_model import Lasso
>>> from sklearn.svm import SVR
>>> from pandas import DataFrame
```

[3] Sapp, S., van der Laan, M. J., & Canny, J. (2014). Subsemble: an ensemble method for combining subset-specific algorithm fits. Journal of Applied Statistics, 41(6), 1247-1259. http://doi.org/10.1080/02664763.2013.864263

```
>>>
>>> X, y = load_boston(True)
>>>
>>> ensemble = SequentialEnsemble(scorer=rmse)
>>>
>>> # Add a subsemble with 10 partitions and 10 folds as first layer
>>> ensemble.add('subset', [SVR(), Lasso()], n_partitions=10, n_splits=10)
>>>
>>> # Add a super learner with 20 folds as second layer
>>> ensemble.add('stack', [SVR(), Lasso()], n_splits=20)
>>>
>>> # Specify a meta estimator
>>> ensemble.add_meta(SVR())
>>>
>>> ensemble.fit(X, y)
>>>
>>> DataFrame(ensemble.scores_)
                   score_mean  score_std
layer-1 j0__lasso   11.792905   2.744788
        j0__svr      9.615539   1.185780
        j1__lasso    7.525038   1.235617
        j1__svr      9.164761   0.896510
        j2__lasso    7.239405   1.821464
        j2__svr      9.965071   1.357993
        j3__lasso    9.590788   1.723333
        j3__svr     11.892205   0.880309
        j4__lasso   12.435838   3.475319
        j4__svr      9.368308   0.769086
        j5__lasso   17.357559   2.645452
        j5__svr     11.921103   1.217075
        j6__lasso    8.889963   1.811024
        j6__svr      9.226893   1.030218
        j7__lasso   12.720208   3.521461
        j7__svr     12.751075   1.760458
        j8__lasso   12.178918   1.229540
        j8__svr     12.878269   1.667963
        j9__lasso    7.269251   1.815074
        j9__svr      9.563657   1.214829
layer-2 lasso        5.660264   2.435897
        svr          8.343091   4.097081
```

Note how each of the two base learners specified got duplicated to each of the 10 partitions, as denotes by the `j[num]_` prefix.

1.5.6 Memory consumption

Memory mapping

When training data is stored in-memory in the parent process, training a ensemble in parallel entails sending the array from the parent process to the subprocess through serialization of the data. Even for moderately sized datasets, this is a time consuming task. Moreover, it creates replicas of the same dataset to be stored in-memory, and so the effective size of the data kept in memory scales with the number of processes used in parallel. For large datasets, this can be catastrophic.

ML-Ensemble overcomes this issue by using memmapping, which allows sub-processes to share memory of the underlying data. Hence, input data need not be serialized and sent to the subprocesses, and as long as no copying takes

place in the sub-process, memory consumption remains constant as the number of sub-processes grows. Hence, ML-Ensemble can remain memory neutral as the number of CPU's in use increase. This last point relies critically on avoiding copying, which may not be possible, see *Gotcha's* for further information.

We can easily illustrate this issue by running a dummy function in parallel that merely holds whatever data it receives from a few seconds before closing. Here, we make use of the `CMLog` monitor that logs the memory (and cpu) usage of the process that instantiated it.

```
>>> import numpy as np
>>> from joblib import Parallel, delayed
>>> from time import sleep, perf_counter
>>> from mlens.utils.utils import CMLog

>>> def hold(arr, s):
...     """Hold an array ``arr`` in memory for ``s`` seconds."""
...     sleep(s)

>>> # Monitor memory usage
>>> cm = CMLog()
>>> cm.monitor()

>>> sleep(3)

>>> # Load an approx. 800MB array into memory
>>> t1 = int(np.floor(perf_counter() - cm._t0) * 10)

>>> array = np.arange(int(1e8)).reshape(int(1e5), int(1e3))

>>> sleep(3)
>>> # Launch 4 sub-process, each holding a copy of the array in memory.
>>> t2 = int(np.floor(perf_counter() - cm._t0) * 10)

>>> Parallel(n_jobs=-1, verbose=100, max_nbytes=None)(
...         delayed(hold)(array, 3)
...         for _ in range(4))

>>> t3 = int(np.floor(perf_counter() - cm._t0) * 10)

>>> # Job done
>>> sleep(3)
>>> cm.collect()
Pickling array (shape=(100000, 1000), dtype=int64).
Pickling array (shape=(100000, 1000), dtype=int64).
[Parallel(n_jobs=-1)]: Done   1 tasks      | elapsed:   17.1s
Pickling array (shape=(100000, 1000), dtype=int64).
[Parallel(n_jobs=-1)]: Done   5 out of   4 | elapsed:   26.3s remaining:   -5.3s
Pickling array (shape=(100000, 1000), dtype=int64).
[Parallel(n_jobs=-1)]: Done   5 out of   4 | elapsed:   36.4s remaining:   -7.3s
[Parallel(n_jobs=-1)]: Done   5 out of   4 | elapsed:   43.8s remaining:   -8.8s
[Parallel(n_jobs=-1)]: Done   4 out of   4 | elapsed:   43.8s finished
```

Notice that the parallel job seems to be doing an awful lot of data serialization. The memory log of the `cm` reveals that peak memory usage is over some three times larger than the original array when 4 cpu's are in use. With such a memory profile, an ensemble would not be very scalable.

Memmapping allows us to overcome these issues for two reaons. First, it entirely overcomes serialization of the input data as processes share memory and hence the subprocesses can access the input arrays directly from the parent process. Second, insofar no copying of the input data takes place, memmapping avoids scaling the data size requirement by the number of processes running. To see this first hand, we can modify the above example to convert the toy array to a memmap and again monitor memory usage.

```
>>> import os
>>> import tempfile
>>> from joblib import load, dump

>>> with tempfile.TemporaryDirectory() as tmpdir:
>>>     f = os.path.join(tmpdir, 'arr.mmap')
>>>     if os.path.exists(f): os.unlink(f)

>>>     cm = CMLog(True)
>>>     cm.monitor()

>>>     sleep(3)

>>>     array = np.arange(int(1e8)).reshape(int(1e5), int(1e3))
>>>     t1 = int(np.floor(perf_counter() - cm._t0) * 10)

>>>     # Now, we dump the array into a memmap in the temporary directory
>>>     dump(array, f)
>>>     array = load(f, mmap_mode='r+')
>>>     t1_d = int(np.floor(perf_counter() - cm._t0) * 10)
```

```
>>>     sleep(3)

>>>     t2 = int(np.floor(perf_counter() - cm._t0) * 10)

>>>     Parallel(n_jobs=-1, verbose=100, max_nbytes=None)(
...              delayed(hold)(array, 3)
...              for _ in range(4))

>>>     t3 = int(np.floor(perf_counter() - cm._t0) * 10)

>>>     sleep(3)

>>>      cm.collect()
[Parallel(n_jobs=-1)]: Done   1 tasks      | elapsed:    3.0s
[Parallel(n_jobs=-1)]: Done   5 out of   4 | elapsed:    3.0s remaining:   -0.6s
[Parallel(n_jobs=-1)]: Done   5 out of   4 | elapsed:    3.0s remaining:   -0.6s
[Parallel(n_jobs=-1)]: Done   5 out of   4 | elapsed:    3.0s remaining:   -0.6s
[Parallel(n_jobs=-1)]: Done   4 out of   4 | elapsed:    3.0s finished
```

Notice first that no pickling is reported in the parallel job; second, the time to completion is no more than the 3 seconds we asked the `hold` function to sleep. In other words, memmaping causes *no* process time overhead. This stands in stark contrast to the previous example, which needed over 40 seconds to complete - an order of magnitude slower. Moreover, inspecting the memory profile, note that memmapping is completely memory neutral. In fact, if we replace the original array with the memmap (as in this example), the memory required to hold the original file can be released and so there is *no* copy of the array kept in the process memory.

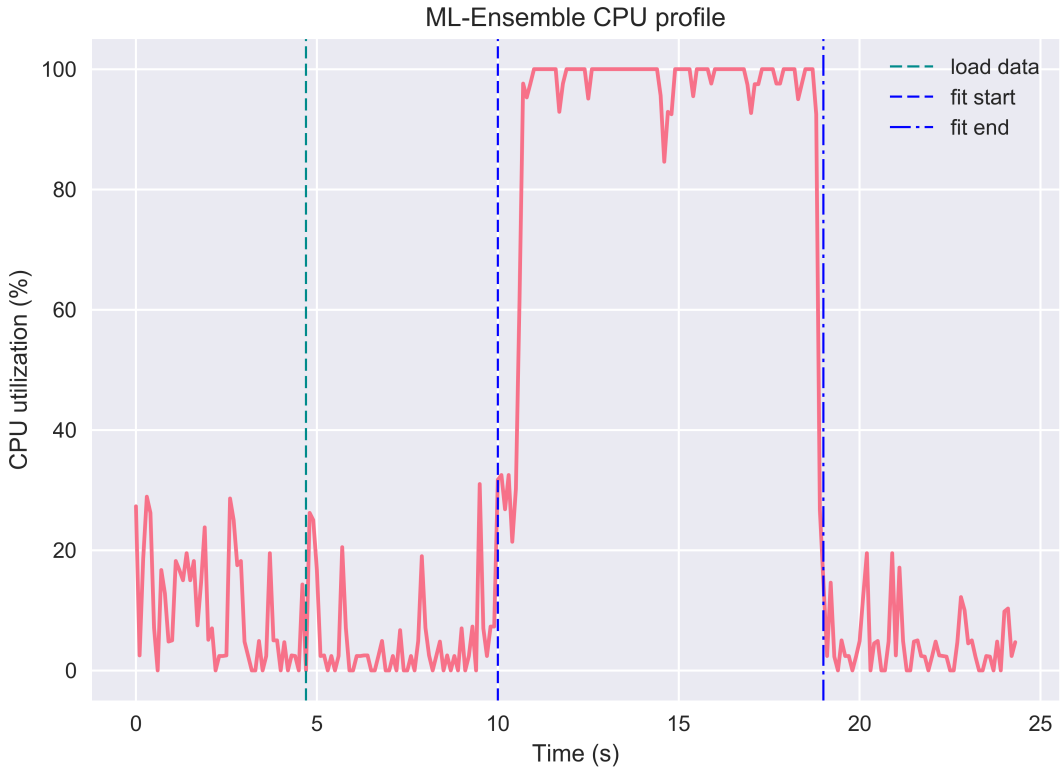
For further details on memmapping in parallel processing, see the joblib package's documentation.

ML-Ensemble memory profiling

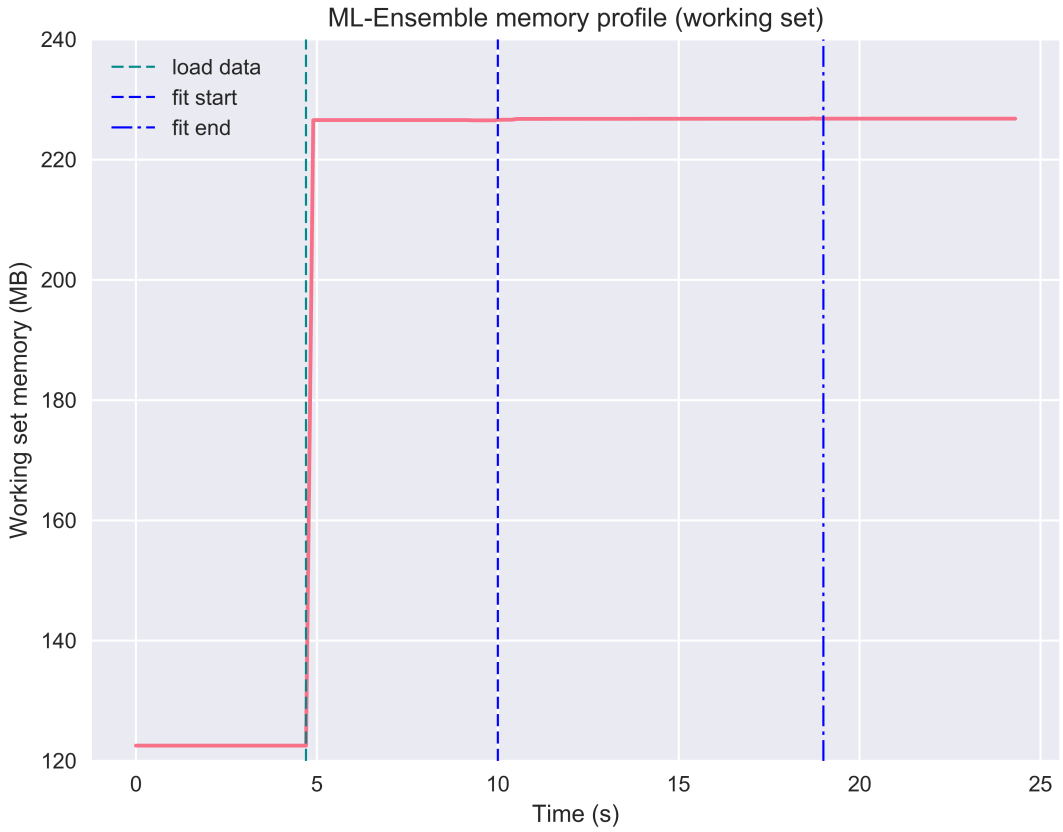
By leveraging memmapping, ML-Ensemble estimators are able to achieve memory neutral parallel processing. In the following example, an ensemble of three linear regression estimators with different preprocessing pipelines are fitted on data comprising 6 million observations and ten features. The following profiling can be run from the package root with the below command:

```
>>> python benchmarks/memory_cpu_profile.py
```

Note that the ensemble leveraged the full capacity of the CPU to fit the ensemble.



And while doing so, memory consumption remained neutral. Note here that becase the input data was first loaded into memory, then passed to the ensemble, the original data stays in memory (the ensemble instance cannot delete objects outside it's scope). To make the ensemble even more memory efficient, a user can specify a path to a csv file or stored numpy array or numpy memmap, in which case no memory will be committed to keeping the original data in memory. See the *Passing file paths as data input* tutorial for more information.

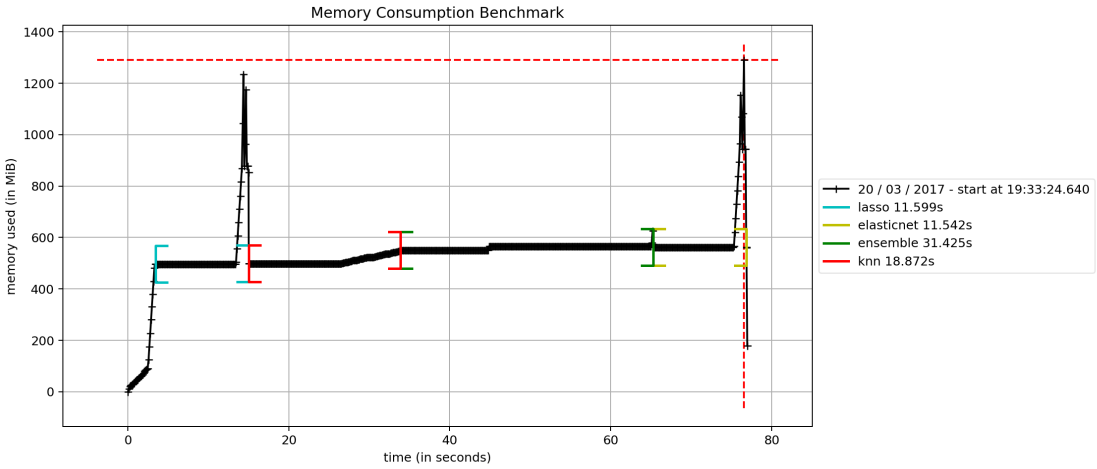


Memory performance benchmark

Finally, we consider how a `SuperLearner` compares in terms of memory consumption against a set of Scikit-learn estimators. This benchmark relies on the mprof package, which can be installed with `pip`. The benchmark compares the `sklearn.linear_model.Lasso`, `sklearn.linear_model.ElasticNet` and the `sklearn.neighbors.KNeighborsRegressor` against an ensemble that uses the former two as the first layer and the latter as a final meta estimator.

```
>>> mprof run friedman_memory.py
>>> mprof plot friedman_memory.py -t "Memory Consumption Benchmark"
mprof: Sampling memory every 0.1s
running as a Python program...

ML-ENSEMBLE

Benchmark of ML-ENSEMBLE memory profile against Scikit-learn estimators.

Data shape: (1000000, 50)

Data size: 400 MB

Fitting LAS... Done | 00:00:01

Fitting KNN... Done | 00:00:08
```

```
Fitting ENS... Done | 00:00:21

Fitting ELN... Done | 00:00:01

Profiling complete. | 00:01:13

Using last profile data.
```



Gotcha's

The above analysis holds under two conditions: (1) no copying of the input data is triggered during slicing the K-folds and (2) the base estimators do not copy the data internally. However memmapping always avoids array serialization between sub-processes which can be significant burden on time consumption.

(1) Because of the structure of numpy's memory model, slicing an array returns a view only if the slice contiguous. In particular, this means that we **cannot** slice a numpy array to retrieve two partitions separated by one or more partitions. Technically, this limitation arises since it breaks the stride patterns numpy arrays relies on to know where find a row. ML-Ensemble can therefore **only** avoid copying training data when the number of folds is 2, in which case the first half is used for training and the latter for predictions. For 3 of more folds, the training set is no longer contiguous and hence slicing the original array triggers advanced indexing, in turn causing a copy of the underlying data to be returned. Being a limitation within numpy, this issue is beyond the control of ML-Ensemble.

Also note that if the data is preprocessed within ML-Ensemble, transformers automatically return copies of the input data (i.e. breaks the link with the memory buffer) and will therefore **always** trigger a copying. In fact, if it does not, transforming the memmapped original data will raise an `OSError` since the memory map of the original data is read-only to avoid corrupting the input.

(2) The user must take not what input requirements are necessary for a Scikit-learn estimator to not copy the data, and ensuring the input array is in the given format. Note that prediction arrays are always dense C-ordered float64 arrays. For instance, several Scikit-learn linear models defaults to copying the input data, Scikit-learn random forests estimators copy the data if it is not Fortran contiguous. Similarly, Scikit-learn SVM models copy data that does not satisfy its particular requirements.

1.5.7 Performance benchmarks

The Friedman Regression Problem 1

The The Friedman Regression Problem 1, as described in[1] and[2], is constructed as follows. Set some sample size m , feature dimensionality n, and noise level e. Then the input data $\mathbf{X}$ and output data $y(\mathbf{X})$ is given by:

$$\mathbf{X} = [X_i]_{i \in \{1,2,\ldots,n\}} \in \mathbb{R}^{m \times n},$$
$$X \sim u[0, 1],$$

$$y(\mathbf{X}) = 10 \sin(\pi X_1 X_2) + 20(X_3 - 0.5)^2 + 10X_4 + 5X_5 + \epsilon,$$

$$\epsilon \sim \mathrm{N}(0, e).$$

Benchmark

The following benchmark uses 10 features and scores a relatively wide selection of Scikit-learn estimators against a specified *SuperLearner*. All estimators are used with default parameter settings. As such, the benchmark does not reflect the best possible score of each estimator, but shows rather how stacking even relatively low-performing estimators can yield superior predictive power. In this case, the Super Learner improves on the best stand-alone estimator by 25%.

```
>>> python friedman_scores.py
Benchmark of ML-ENSEMBLE against Scikit-learn estimators on the friedman1 dataset.

Scoring metric: Root Mean Squared Error.

Available CPUs: 4

Ensemble architecture
Num layers: 2
layer-1 | Min Max Scaling - Estimators: ['svr'].
layer-1 | Standard Scaling - Estimators: ['elasticnet', 'lasso', 'kneighborsregressor
↪'].
layer-1 | No Preprocessing - Estimators: ['randomforestregressor',
↪'gradientboostingregressor'].
layer-2 | (meta) GradientBoostingRegressor

Benchmark estimators: GBM KNN Kernel Ridge Lasso Random Forest SVR Elastic-Net

Data
Features: 10
Training set sizes: from 2000 to 20000 with step size 2000.

SCORES
  size | Ensemble |      GBM |      KNN | Kern Rid |    Lasso | Random F |      SVR |␣
↪   elNet |
  2000 |     0.83 |     0.92 |     2.26 |     2.42 |     3.13 |     1.61 |     2.32 |␣
↪    3.18 |
  4000 |     0.75 |     0.91 |     2.11 |     2.49 |     3.13 |     1.39 |     2.31 |␣
↪    3.16 |
  6000 |     0.66 |     0.83 |     2.02 |     2.43 |     3.21 |     1.29 |     2.18 |␣
↪    3.25 |
```

[1] J. Friedman, "Multivariate adaptive regression splines", The Annals of Statistics 19 (1), pages 1-67, 1991.
[2] L. Breiman, "Bagging predictors", Machine Learning 24, pages 123-140, 1996.

```
  8000 |     0.66 |     0.84 |     1.95 |     2.43 |     3.19 |     1.24 |     2.09 |
↪    3.24 |
 10000 |     0.62 |     0.79 |     1.90 |     2.46 |     3.17 |     1.16 |     2.03 |
↪    3.21 |
 12000 |     0.68 |     0.86 |     1.84 |     2.46 |     3.16 |     1.10 |     1.97 |
↪    3.21 |
 14000 |     0.59 |     0.75 |     1.78 |     2.45 |     3.15 |     1.05 |     1.92 |
↪    3.20 |
 16000 |     0.62 |     0.80 |     1.76 |     2.45 |     3.15 |     1.02 |     1.87 |
↪    3.19 |
 18000 |     0.59 |     0.79 |     1.73 |     2.43 |     3.12 |     1.01 |     1.83 |
↪    3.17 |
 20000 |     0.56 |     0.73 |     1.70 |     2.42 |     4.87 |     0.99 |     1.81 |
↪    4.75 |

FIT TIMES
  size | Ensemble |      GBM |      KNN | Kern Rid |    Lasso | Random F |      SVR |
↪   elNet |
  2000 |     0:01 |     0:00 |     0:00 |     0:00 |     0:00 |     0:00 |     0:00 |
↪    0:00 |
  4000 |     0:02 |     0:00 |     0:00 |     0:00 |     0:00 |     0:00 |     0:00 |
↪    0:00 |
  6000 |     0:03 |     0:00 |     0:00 |     0:01 |     0:00 |     0:00 |     0:01 |
↪    0:00 |
  8000 |     0:04 |     0:00 |     0:00 |     0:04 |     0:00 |     0:00 |     0:02 |
↪    0:00 |
 10000 |     0:06 |     0:01 |     0:00 |     0:08 |     0:00 |     0:00 |     0:03 |
↪    0:00 |
 12000 |     0:08 |     0:01 |     0:00 |     0:12 |     0:00 |     0:00 |     0:04 |
↪    0:00 |
 14000 |     0:10 |     0:01 |     0:00 |     0:20 |     0:00 |     0:00 |     0:06 |
↪    0:00 |
 16000 |     0:13 |     0:02 |     0:00 |     0:34 |     0:00 |     0:00 |     0:08 |
↪    0:00 |
 18000 |     0:17 |     0:02 |     0:00 |     0:47 |     0:00 |     0:00 |     0:10 |
↪    0:00 |
 20000 |     0:20 |     0:02 |     0:00 |     1:20 |     0:00 |     0:00 |     0:13 |
↪    0:00 |
```

References

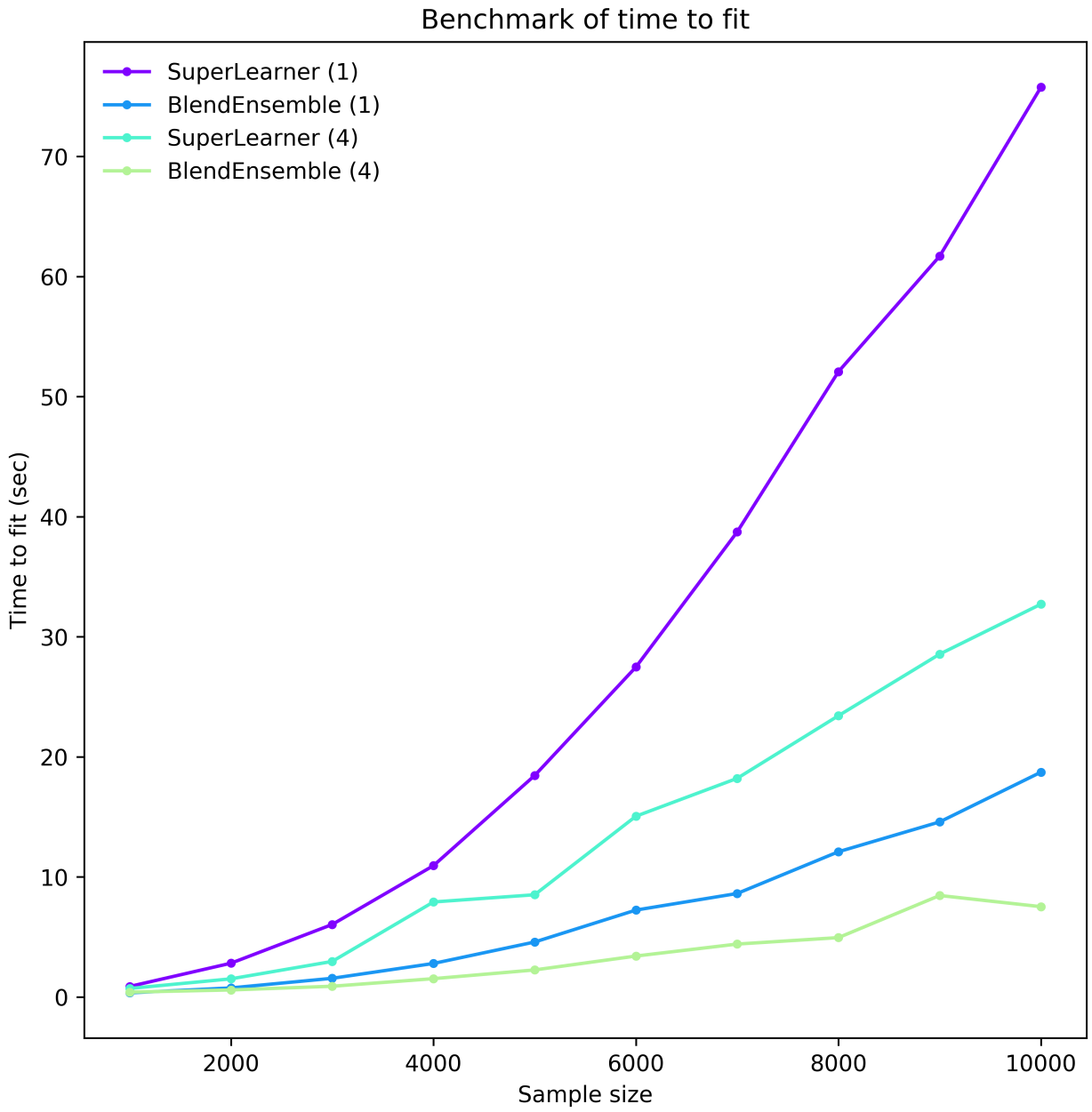
1.5.8 Scale benchmarks

The *Single process vs multi-process* benchmark compares how running ensembles on a single process fares against running them on multiple processes.

The *Ensemble comparison* benchmark compares ensemble classes in terms of time to fit and predictive power as data scales.

Single process vs multi-process

We compare the time to fit the `SuperLearner` and the `BlendEnsemble` when run on a single process and when run on four processes. The ensembles have four SVR base estimators and an SVR as final meta learner. Hence, while the single-processed ensembles need to fit 5 SVR models consecutively, the multiprocessed ensembles need only the time equivalent to fit 2 consecutively. As the figure below shows, there are clear benefits to multi-processing.



To replicate the benchmark, in the `mlens` benchmark folder, execute:

```
>>> python scale_cpu.py

ML-ENSEMBLE

Threading performance test for data set dimensioned up to (10000, 50)
Available CPUs: 4

Ensemble architecture
Num layers: 2
Fit per base layer estimator: 2 + 1
layer-1 | Estimators: ['svr-1', 'svr-2', 'svr-3', 'svr-4'].
layer-2 | Meta Estimator: svr
```

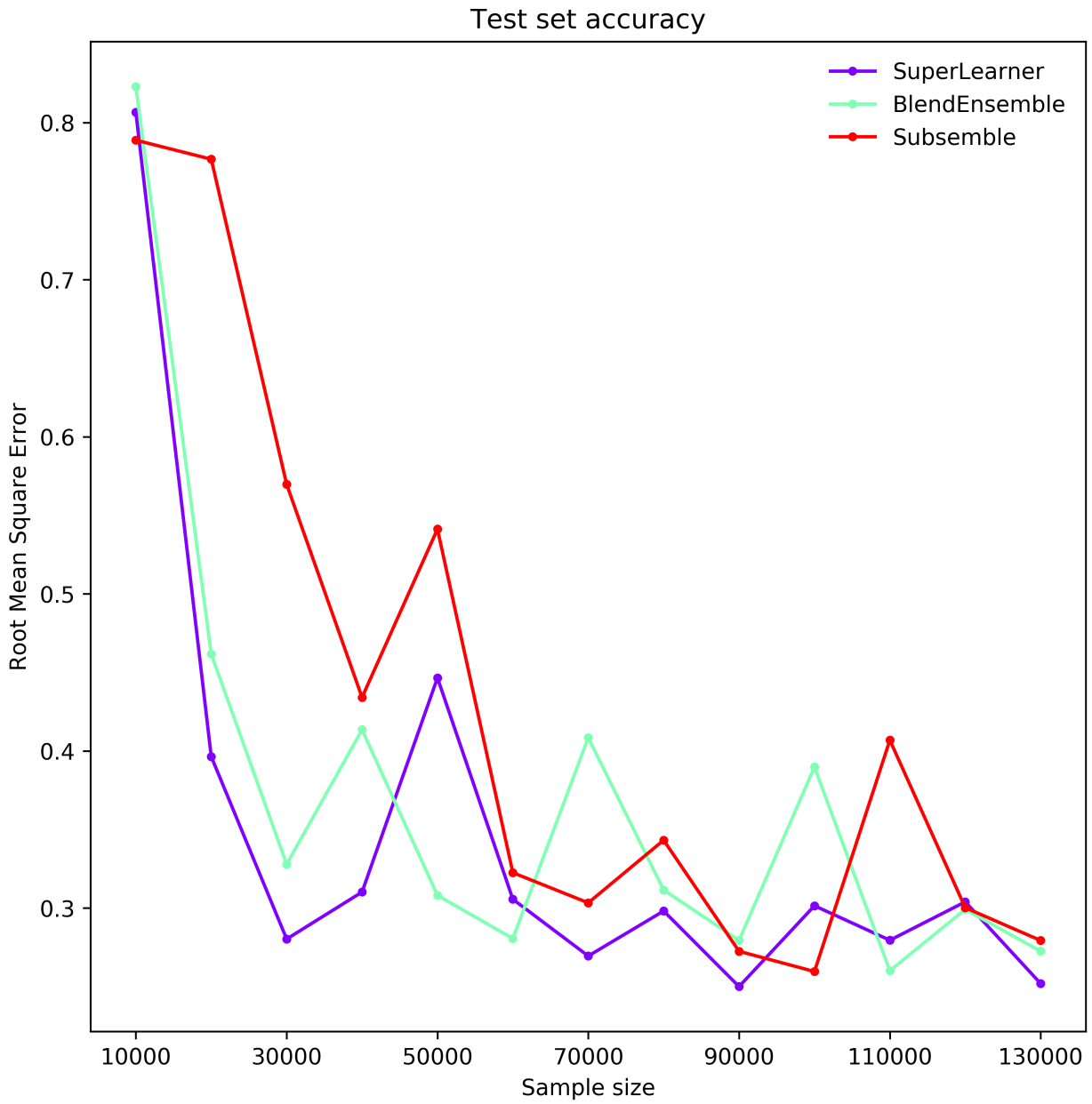
```
FIT TIMES
samples
   1000 SuperLearner (1) :   0.88s | BlendEnsemble (1) :   0.35s |
   1000 SuperLearner (4) :   0.71s | BlendEnsemble (4) :   0.41s |

   2000 SuperLearner (1) :   2.82s | BlendEnsemble (1) :   0.76s |
   2000 SuperLearner (4) :   1.51s | BlendEnsemble (4) :   0.59s |

   3000 SuperLearner (1) :   6.04s | BlendEnsemble (1) :   1.56s |
   3000 SuperLearner (4) :   2.96s | BlendEnsemble (4) :   0.90s |

   4000 SuperLearner (1) :  10.94s | BlendEnsemble (1) :   2.79s |
   4000 SuperLearner (4) :   7.92s | BlendEnsemble (4) :   1.53s |

   5000 SuperLearner (1) :  18.45s | BlendEnsemble (1) :   4.58s |
   5000 SuperLearner (4) :   8.52s | BlendEnsemble (4) :   2.26s |

   6000 SuperLearner (1) :  27.48s | BlendEnsemble (1) :   7.24s |
   6000 SuperLearner (4) :  15.06s | BlendEnsemble (4) :   3.41s |

   7000 SuperLearner (1) :  38.73s | BlendEnsemble (1) :   8.62s |
   7000 SuperLearner (4) :  18.21s | BlendEnsemble (4) :   4.41s |

   8000 SuperLearner (1) :  52.08s | BlendEnsemble (1) :  12.10s |
   8000 SuperLearner (4) :  23.43s | BlendEnsemble (4) :   4.95s |

   9000 SuperLearner (1) :  61.70s | BlendEnsemble (1) :  14.58s |
   9000 SuperLearner (4) :  28.55s | BlendEnsemble (4) :   8.45s |

  10000 SuperLearner (1) :  75.76s | BlendEnsemble (1) :  18.72s |
  10000 SuperLearner (4) :  32.71s | BlendEnsemble (4) :   7.52s |

Benchmark done | 00:09:00
```

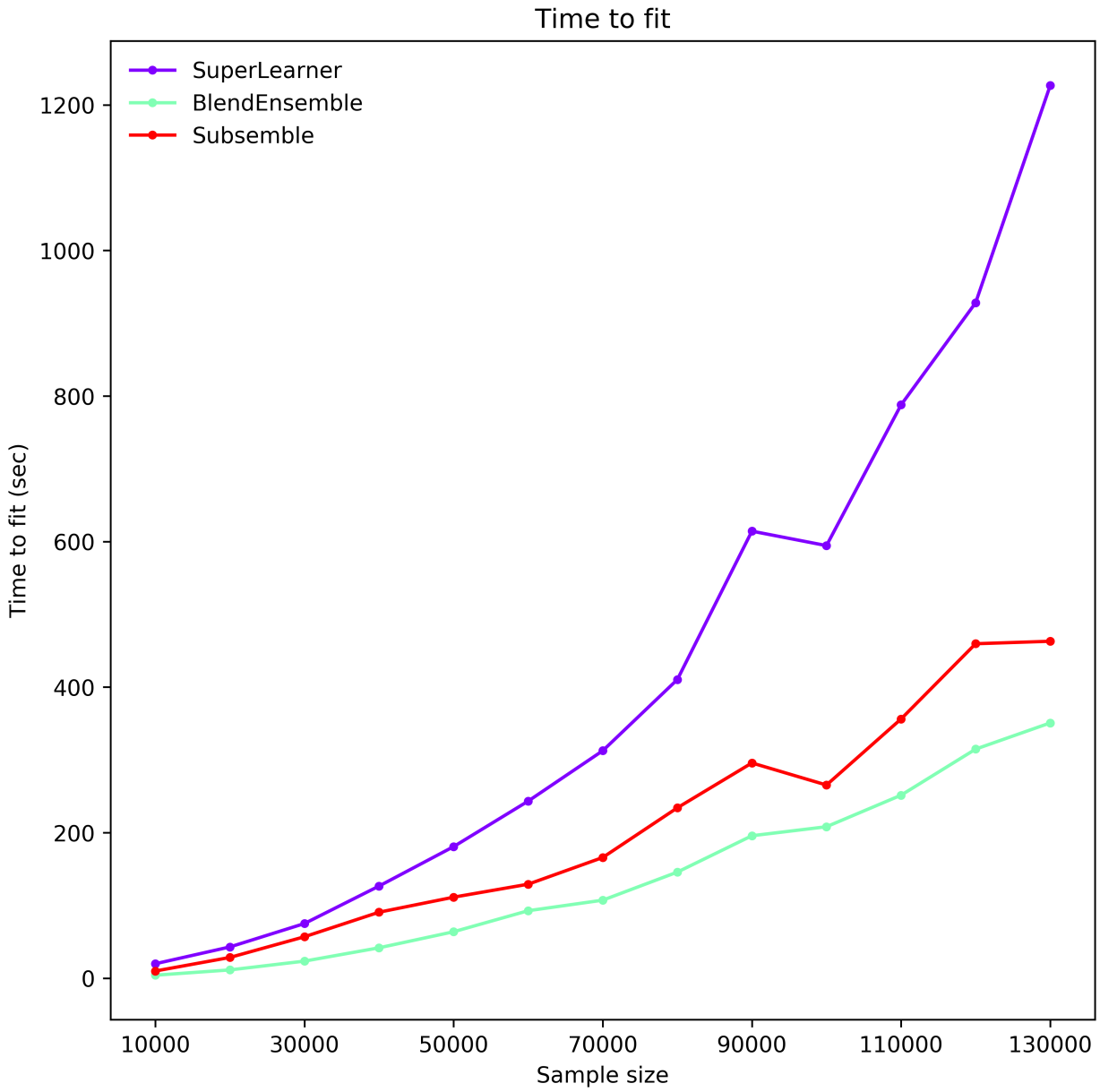
Ensemble comparison

We compare the time to fit a Super Learner, Subsemble and Blend ensemble when run on four processes as data scales from 20 000 to 260 000 observations with 20 dense real valued features.

Each ensemble has the same set of base learners and meta learners, all initiated at standard parameter settings. Each model is fitted on half the observations and predict the other half. The data is generated as per the Friedman 1 process (see *Performance benchmarks*).

The super learner tends to give best performance (rmse), but generally all classes achieve similar accuracy scores. However, the super learner (with 2 folds) takes more than twice as long to fit than a subsemble (with 3 partitions and 2 folds on each partition), and up to three times as long as the Blend ensemble (with 50% split). The subsemble tends to perform better than the blend ensemble after 70000 observations and has a similar fit time. In fact, it can be made more time efficient if the number of partitions is increased. This can significantly impact optimal meta learner parameter settings and overall performance.


Test set accuracy
SuperLearner
BlendEnsemble
Subsemble
Root Mean Square Error
0.8
0.7
0.6
0.5
0.4
0.3
10000
30000
50000
70000
90000
110000
130000
Sample size



To replicate the benchmark, in the `mlens` benchmark folder, execute:

```
>>> python scale_ens.py

ML-ENSEMBLE

Ensemble scale benchmark for datadimensioned up to (250000, 20)
Available CPUs: 4

Ensemble architecture
Num layers: 2
layer-1 | Estimators: ['svr', 'randomforestregressor', 'gradientboostingregressor',
↪'lasso', 'mlpregressor'].
layer-2 | Meta Estimator: lasso

SCORES (TIME TO FIT)
```

```
Sample size
      20000 SuperLearner : 0.807 ( 19.83s) | BlendEnsemble : 0.823 (  4.09s) |␣
↪Subsemble : 0.789 (  9.84s) |
      40000 SuperLearner : 0.396 ( 42.94s) | BlendEnsemble : 0.462 ( 11.37s) |␣
↪Subsemble : 0.777 ( 28.49s) |
      60000 SuperLearner : 0.280 ( 75.08s) | BlendEnsemble : 0.328 ( 23.43s) |␣
↪Subsemble : 0.570 ( 56.93s) |
      80000 SuperLearner : 0.310 (126.59s) | BlendEnsemble : 0.414 ( 41.75s) |␣
↪Subsemble : 0.434 ( 90.66s) |
     100000 SuperLearner : 0.447 (180.77s) | BlendEnsemble : 0.308 ( 63.80s) |␣
↪Subsemble : 0.541 (111.31s) |
     120000 SuperLearner : 0.306 (243.34s) | BlendEnsemble : 0.281 ( 92.71s) |␣
↪Subsemble : 0.323 (129.15s) |
     140000 SuperLearner : 0.269 (312.58s) | BlendEnsemble : 0.408 (107.19s) |␣
↪Subsemble : 0.303 (165.86s) |
     160000 SuperLearner : 0.298 (410.33s) | BlendEnsemble : 0.312 (145.76s) |␣
↪Subsemble : 0.343 (234.12s) |
     180000 SuperLearner : 0.250 (614.27s) | BlendEnsemble : 0.279 (195.74s) |␣
↪Subsemble : 0.272 (295.76s) |
     200000 SuperLearner : 0.301 (594.41s) | BlendEnsemble : 0.390 (208.11s) |␣
↪Subsemble : 0.260 (265.42s) |
     220000 SuperLearner : 0.280 (787.79s) | BlendEnsemble : 0.260 (251.45s) |␣
↪Subsemble : 0.407 (356.17s) |
     240000 SuperLearner : 0.304 (928.15s) | BlendEnsemble : 0.299 (314.76s) |␣
↪Subsemble : 0.300 (459.59s) |
     260000 SuperLearner : 0.252 (1226.66s) | BlendEnsemble : 0.273 (350.77s) |␣
↪Subsemble : 0.279 (462.97s) |
Benchmark done | 04:20:34
```

1.5.9 Hacking ML-Ensemble

ML-Ensemble implements a modular design that allows straightforward development of new ensemble classes. The backend is agnostic to the type of ensemble it is being asked to perform computation on, and only at the moment of computation will ensemble-specific code be needed. To implement a new ensemble type, three objects are needed:

1. **An cross-validation strategy. This amounts to implementing an** indexer class. See *current indexers* for examples.
2. **An estimation engine. This is the actual class that will run the** estimation. The `BaseEstimator` class implements most of the heavy lifting, and unless special-purpose fit and/or predict procedures are required, the only thing needed is a method for indexing the base learners to each new features generated by the cross-validation strategy. See *current estimation engines* for examples.
3. **A front-end API. These typically only implements a constructor and an** `add` method. The `add` method specifies the indexer to use and parser keyword arguments. It is also adviced to differentiate between hidden layers and the meta layer, where cross-validation is not desired.

1.5.10 Troubleshooting

Here we collect a set of subtle potential issues and limitations that may explain odd behavior that you have encountered. Feel free to reach out if your problem is not addressed here.

Bad interaction with third-party packages

ML-Ensemble is designed to work with any estimator that implements a minimal API, and is specifically unit tested to work with Scikit-learn. When using estimators from other libraries, it can happen that the estimation stalls and fails to complete. A clear sign of this is if there is no python process with high CPU usage.

Due to how Python runs processes in parallel, child workers can receive a corrupted thread state that causes the worker to try to acquire more threads than are available, resulting in a deadlock. If this happens, raise an issue at the Github repository. There are a few things to try that might alleviate the problem:

1. ensure that all estimators in the ensemble or evaluator has `n_jobs` or `nthread` equal to `1`,
2. change the `backend` parameter to either `threading` or `multiprocessing` depending on what the current setting is,
3. try using `multiprocessing` together with a fork method (see *Global configurations*).

For more information on this issue see the Scikit-learn FAQ.

Array copying during fitting

When the number of folds is greater than 2, it is not possible to slice the full data in such a way as to return a view of that array (i.e. without copying any data). Hence for fold numbers larger than 2, each subprocess will in fact trigger a copy of the training data (which can be from 67% to 99% of the full data size). A possible alleviation to this problem is to memmap the required slices before estimation to avoid creating new copies in each subprocess. However this will induce the equivalent of several copies of the underlying data to be persisted to disk and may instead lead to the issue remaining as a disk-bound issue. Since elementary diagnostics suggest that for data sizes where memory becomes a constraining factor, increasing the number of folds beyond 2 does not significantly impact performance and at this time of writing this is the suggested approach. For further information on avoiding copying data during estimation, see *Memory consumption*.

File permissions on Windows

During ensemble estimation, ML-Ensemble will create a temporary directory and populate it with training data and predictions, along with pickled estimators and transformers. Each subprocess is given an container object that points to the objects in the directory, and once the estimation is done the temporary directory is cleaned and removed. The native python execution of the termination typically fails due to how Windows gives read and write permission between processes. To overcome this, ML-Ensemble runs an explicit shell command (`rmdir -s -q dir`) that forcibly removes the cache. Current testing on development machines indicates this exception handling is successful and Windows users should not expect any issues. If however you do notice memory performance issues, create an issue at the issue tracker.

1.5.11 API

ML-Ensemble estimators behave identically to Scikit-learn estimators, with one main difference: to properly instantiate an ensemble, at least on layer, and if applicable a meta estimator, must be added to the ensemble. Otherwise, there is no ensemble to estimate. The difference can be summarized as follows.

```
# sklearn API
estimator = Estimator()
estimator.fit(X, y)

# mlens API
ensemble = Ensemble().add(list_of_estimators).add_meta(estimator)
ensemble.fit(X, y)
```

Ensemble estimators

SuperLearner([folds, shuffle, random_state, ...])	Super Learner class.
Subsemble([partitions, partition_estimator, ...])	Subsemble class.
BlendEnsemble([test_size, shuffle, ...])	Blend Ensemble class.
SequentialEnsemble([shuffle, random_state, ...])	Sequential Ensemble class.

Model Selection

Evaluator(scorer[, cv, shuffle, ...])	Model selection across several estimators and preprocessing pipelines.

Preprocessing

EnsembleTransformer([shuffle, random_state, ...])	Ensemble Transformer class.
Subset([subset])	Select a subset of features.

Visualization

corrmat(corr[, figsize, annotate, inflate, ...])	Function for generating color-coded correlation triangle.
clustered_corrmap(corr, cls[, ...])	Function for plotting a clustered correlation heatmap.
corr_X_y(X, y[, top, figsize, fontsize, ...])	Function for plotting input feature correlations with output.
pca_plot(X, estimator[, y, cmap, figsize, ...])	Function to plot a PCA analysis of 1, 2, or 3 dims.
pca_comp_plot(X[, y, figsize, title, ...])	Function for comparing PCA analysis.
exp_var_plot(X, estimator[, figsize, ...])	Function to plot the explained variance using PCA.

1.5.12 For developers

The following base classes are good starting points for building new ensembles. You may want to study the source code directly.

Indexers

IdTrain([size])	Container to identify training set.
BlendIndex([test_size, train_size, X, ...])	Indexer that generates two non-overlapping subsets of `X`.
FoldIndex([n_splits, X, raise_on_exception])	Indexer that generates the full size of `X`.
SubsetIndex([n_partitions, n_splits, X, ...])	Subsample index generator.
FullIndex([X])	Vacuous indexer to be used with final layers.
ClusteredSubsetIndex(estimator[, ...])	Clustered Subsample index generator.

Estimation routines

ParallelProcessing(caller)	Parallel processing engine.
ParallelEvaluation(caller)	Parallel cross-validation engine.

Continued on next page

Table 1.6 – continued from previous page

Stacker(job, layer)	Stacked fit sub-process class.
Blender(job, layer)	Blended fit sub-process class.
SubStacker(job, layer)	Stacked subset fit sub-process class.
SingleRun(job, layer)	Single run fit sub-process class.
Evaluation(evaluator)	Evaluation engine.
BaseEstimator(layer)	Base class for estimating a layer in parallel.

1.5.13 Global configurations

ML-Ensemble allows a set of low-level global configurations to tailor the behavior of classes during estimation. Every variable is accessible through `mlens.config`. Alternatively, all variables can be set as global environmental variables, where the exported variable name is `MLENS_[VARNAME]`.

- **`mlens.config.BACKEND`** configures the global default backend during parallelized estimation. Default is `'threading'`. Options are `'multiprocessing'` and `'forkserver'`. See joblib for further information. Alter with the `set_backend` function.
- **`mlens.config.DTYPE`** determines the default dtype of numpy arrays created during estimation; in particular, the prediction matrices of each intermediate layer. Default is `numpy.float32`. Alter with the `set_backend` function.
- **`mlens.config.TMPDIR`** The directory where temporary folders are created during estimation. Default uses the tempfile function `gettempdir()`. Alter with the `set_backend` function.
- **`mlens.config.START_METHOD`** The method used by the job manager to generate a new job. ML-Ensemble defaults to `forkserver``on Unix with Python 3.4+, and ``spawn` on windows. For older Python versions, the default is `fork`. This method has the least overhead, but it can cause issues with third-party software. See *Bad interaction with third-party packages* for details. Set this variable with the `set_start_method` function.

1.5.14 Licence

MIT License

Copyright (c) 2017 Sebastian Flennerhag

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

1.5.15 Change log

- **04/2017: Release of version 0.1.3**

  - Initial stable version released.
- **07/2017: Release of version 0.1.4**
  - Prediction array dtype option (default=float32)
  - *Feature propagation*
  - *Clustered subsemble partitioning*
  - No memmaps passed to estimators (only ndarray views)
  - Global configuration (mlens.config)
  - Scoring exception handling
- **07/2017: Release of version 0.1.5**
  - Possible to set environmental variables
  - `spawn` as default start method for parallel jobs (w. multiprocessing)
  - Possible to specify `y` as partition input in *Clustered subsemble partitioning*
  - Minor bug fixes
  - Refactored backend for streamlined front-end feature development
- **07/2017 Release of version 0.1.5.1 and 0.1.5.2**
  - Bug fixes
  - `clear_cache` function to check for residual caches. Safeguard against old caches not being killed.
- **08/2017 Release of version 0.1.6**
  - Propagate sparse input features
  - On the fly prediction array generation
  - Threading as default backend, `fork` as default fork method
  - Bug fixes

1.5.16 mlens

mlens package

Subpackages

mlens.base package

Submodules

mlens.base.id_train module

ML-ENSEMBLE

author Sebastian Flennerhag

copyright 2017

licence MIT

Class for identifying a training set after an estimator has been fitted. Used for determining whether a *predict* or *transform* method should use cross validation to create predictions, or estimators fitted on full training data.

class mlens.base.id_train.**IdTrain**(*size=10*)

Bases: mlens.externals.sklearn.base.BaseEstimator

Container to identify training set.

Samples a random subset from set passed to the *fit* method, to allow identification of the training set in a *transform* or *predict* method.

Parameters **size** (*int*) – size to sample. A random subset of size [size, size] will be stored in the instance.

fit(*X*)

Sample a training set.

Parameters **X** (*array-like*) – training set to sample observations from.

Returns **self** – fitted instance with stored sample.

Return type obj

is_train(*X*)

Check if an array is the training set.

Parameters **X** (*array-like*) – training set to sample observations from.

Returns **self** – fitted instance with stored sample.

Return type obj

mlens.base.id_train.**permutation**(*x*)

Randomly permute a sequence, or return a permuted range.

If *x* is a multi-dimensional array, it is only shuffled along its first index.

Parameters **x** (*int or array_like*) – If *x* is an integer, randomly permute `np.arange(x)`. If *x* is an array, make a copy and shuffle the elements randomly.

Returns **out** – Permuted sequence or array range.

Return type ndarray

Examples

```
>>> np.random.permutation(10)
array([1, 7, 4, 3, 0, 9, 2, 5, 8, 6])
```

```
>>> np.random.permutation([1, 4, 9, 12, 15])
array([15,  1,  9,  4, 12])
```

```
>>> arr = np.arange(9).reshape((3, 3))
>>> np.random.permutation(arr)
array([[6, 7, 8],
       [0, 1, 2],
       [3, 4, 5]])
```

mlens.base.indexer module

ML-ENSEMBLE

author Sebastian Flennerhag

copyright 2017

licence MIT

Classes for partitioning training data.

class mlens.base.indexer.**BaseIndex**
Bases: object

Base Index class.

Specification of indexer-wide methods and attributes that we can always expect to find in any indexer. Helps to provide a uniform interface during parallel estimation.

fit(*X*, *y=None*, *job=None*)
Method for storing array data.

Parameters

- **X** (*array-like of shape [n_samples, optional]*) – array to _collect dimension data from.
- **y** (*array-like, optional*) – label data
- **job** (*str, optional*) – optional job type data

Returns indexer with stores sample size data.

Return type instance

Notes

Fitting an indexer stores nothing that points to the array or memmap `X`. Only the `shape` attribute of `X` is called.

generate(*X=None*, *as_array=False*)
Front-end generator method.

Generator for training and test set indices based on the generator specification in `_gen_indicies`.

Parameters

- **X** (*array-like, optional*) – If instance has not been fitted, the training set `X` must be passed to the `generate` method, which will call `fit` before proceeding. If already fitted, `X` can be omitted.
- **as_array** (*bool (default = False)*) – whether to return train and test indices as a pair of tuple(s) or numpy arrays. If the returned tuples are singular they can be used on an array X with standard slicing syntax (`X[start:stop]`), but if a list of tuples is returned slicing `X` properly requires first building a list or array of index numbers from the list of tuples. This can be achieved either by setting `as_array` to `True`, or running

```
for train_tup, test_tup in indexer.generate():
    train_idx = \
        np.hstack([np.arange(t0, t1) for t0, t1 in train_tup])
```

when slicing is required.

class mlens.base.indexer.**BlendIndex**(*test_size=0.5*, *train_size=None*, *X=None*, *raise_on_exception=True*)

Bases: *mlens.base.indexer.BaseIndex*

Indexer that generates two non-overlapping subsets of X.

Iterator that generates one training fold and one test fold that are non-overlapping and that may or may not partition all of X depending on the user's specification.

BlendIndex creates a singleton generator (has on iteration) that yields two tuples of `(start, stop)` integers that can be used for numpy array slicing (i.e. `X[stop:start]`). If a full array index is desired this can easily be achieved with:

```
for train_tup, test_tup in self.generate():
    train_slice = numpy.hstack([numpy.arange(t0, t1) for t0, t1 in
                                train_tup])

    test_slice = numpy.hstack([numpy.arange(t0, t1) for t0, t1 in
                               test_tup])
```

Parameters

- **test_size** (*int or float (default = 0.5)*) – Size of the test set. If `float`, assumed to be proportion of full data set.
- **train_size** (*int or float, optional*) – Size of test set. If not specified (i.e. `train_size = None`, train_size is equal to `n_samples - test_size`. If `float`, assumed to be a proportion of full data set. If `train_size` + `test_size` amount to less than the observations in the full data set, a subset of specified size will be used.
- **X** (*array-like of shape [n_samples,] , optional*) – the training set to partition. The training label array is also, accepted, as only the first dimension is used. If `X` is not passed at instantiation, the `fit` method must be called before `generate`, or `X` must be passed as an argument of `generate`.
- **raise_on_exception** (*bool (default = True)*) – whether to warn on suspicious slices or raise an error.

See also:

FoldIndex, *SubsetIndex*

Examples

Selecting an absolute test size, with train size as the remainder

```
>>> import numpy as np
>>> from mlens.base.indexer import BlendIndex
>>> X = np.arange(8)
>>> idx = BlendIndex(3, rebase=True)
>>> print('Test size: 3')
>>> for tri, tei in idx.generate(X):
...     print('TEST (idx | array): (%i, %i) | %r ' % (tei[0], tei[1],
...                                                   X[tei[0]:tei[1]]))
...     print('TRAIN (idx | array): (%i, %i) | %r ' % (tri[0], tri[1],
...                                                    X[tri[0]:tri[1]]))
Test size: 3
```

```
TEST (idx | array): (5, 8) | array([5, 6, 7])
TRAIN (idx | array): (0, 5) | array([0, 1, 2, 3, 4])
```

Selecting a test and train size less than the total

```
>>> import numpy as np
>>> from mlens.base.indexer import BlendIndex
>>> X = np.arange(8)
>>> idx = BlendIndex(3, 4, X)
>>> print('Test size: 3')
>>> print('Train size: 4')
>>> for tri, tei in idx.generate(X):
...     print('TEST (idx | array): (%i, %i) | %r ' % (tei[0], tei[1],
...                                                   X[tei[0]:tei[1]]))
...     print('TRAIN (idx | array): (%i, %i) | %r ' % (tri[0], tri[1],
...                                                    X[tri[0]:tri[1]]))
Test size: 3
Train size: 4
TEST (idx | array): (4, 7) | array([4, 5, 6])
TRAIN (idx | array): (0, 4) | array([0, 1, 2, 3])
```

Selecting a percentage of observations as test and train set

```
>>> import numpy as np
>>> from mlens.base.indexer import BlendIndex
>>> X = np.arange(8)
>>> idx = BlendIndex(0.25, 0.45, X)
>>> print('Test size: 25% * 8 = 2')
>>> print('Train size: 45% * 8 < 4 -> 3')
>>> for tri, tei in idx.generate(X):
...     print('TEST (idx | array): (%i, %i) | %r ' % (tei[0], tei[1],
...                                                   X[tei[0]:tei[1]]))
...     print('TRAIN (idx | array): (%i, %i) | %r ' % (tri[0], tri[1],
...                                                    X[tri[0]:tri[1]]))
Test size: 25% * 8 = 2
Train size: 50% * 8 < 4 ->
TEST (idx | array): (3, 5) | array([[3, 4]])
TRAIN (idx | array): (0, 3) | array([[0, 1, 2]])
```

Rebasing the test set to be 0-indexed

```
>>> import numpy as np
>>> from mlens.base.indexer import BlendIndex
>>> X = np.arange(8)
>>> idx = BlendIndex(3, rebase=True)
>>> print('Test size: 3')
>>> for tri, tei in idx.generate(X):
...     print('TEST tuple: (%i, %i) | array: %r' % (tei[0], tei[1],
...                                                 np.arange(tei[0],
...                                                           tei[1])))
Test size: 3
TEST tuple: (0, 3) | array: array([0, 1, 2])
```

fit (*X*, *y=None*, *job=None*)
Method for storing array data.

Parameters

- **X** (*array-like of shape [n_samples, optional]*) – array to _collect dimension data from.
- **y** (*None*) – for compatibility
- **job** (*None*) – for compatibility

Returns indexer with stores sample size data.

Return type instance

class mlens.base.indexer.**ClusteredSubsetIndex**(*estimator*, *n_partitions=2*, *n_splits=2*, *X=None*, *y=None*, *fit_estimator=True*, *attr='predict'*, *partition_on='X'*, *raise_on_exception=True*)

Bases: *mlens.base.indexer.BaseIndex*

Clustered Subsample index generator.

Generates cross-validation folds according used to create J partitions of the data and v folds on each partition according to as per[1]:

1. Split X into J partitions
2. For each partition:
   (a) For each fold v, create train index of all idx not in v
   (b) Concatenate all the fold v indices into a test index for fold v that spans all partitions

Setting J = 1 is equivalent to the FullIndexer, which returns standard K-Fold train and test set indices.

ClusteredSubsetIndex uses a user-provided estimator to partition the data, in contrast to the *SubsetIndex* generator, which partitions data into randomly into equal sizes.

See also:

FoldIndex, *BlendIndex*, *SubsetIndex*

References

Parameters

- **estimator** (*instance*) – Estimator to use for clustering.
- **n_partitions** (*int*) – Number of partitions the estimator will create.
- **n_splits** (*int (default = 2)*) – Number of folds to create in each partition. n_splits can not be 1 if n_partition > 1. Note that if n_splits = 1, both the train and test set will index the full data.
- **fit_estimator** (*bool (default = True)*) – whether to fit the estimator separately before generating labels.
- **attr** (*str (default = 'predict')*) – the attribute to use for generating cluster membership labels.
- **X** (*array-like of shape [n_samples,] , optional*) – the training set to partition. The training label array is also, accepted, as only the first dimension is used. If X is not passed at instantiation, the fit method must be called before generate, or X must be passed as an argument of generate.

[1] Sapp, S., van der Laan, M. J., & Canny, J. (2014). Subsemble: an ensemble method for combining subset-specific algorithm fits. Journal of Applied Statistics, 41(6), 1247-1259. http://doi.org/10.1080/02664763.2013.864263

- **raise_on_exception** (*bool (default = True)*) – whether to warn on suspicious slices or raise an error.

Examples

```
>>> import numpy as np
>>> from sklearn.cluster import KMeans
>>> from mlens.base.indexer import ClusteredSubsetIndex
>>>
>>> km = KMeans(3, random_state=0)
>>> X = np.arange(12).reshape(-1, 1); np.random.shuffle(X)
>>> print("Data: {}".format(X.ravel()))
>>>
>>> s = ClusteredSubsetIndex(km)
>>> s.fit(X)
>>>
>>> P = s.estimator.predict(X)
>>> print("cluster labels: {}".format(P))
>>>
>>> for j, i in enumerate(s.partition(as_array=True)):
...     print("partition ({}) index: {}, cluster labels: {}".format(i, j + 1,␣
↪P[i]))
>>>
>>> for i in s.generate(as_array=True):
...     print("train fold index: {}, cluster labels: {}".format(i[0], P[i[0]]))
Data: [ 8  7  5  2  4 10 11  1  3  6  9  0]
cluster labels: [0 2 2 1 2 0 0 1 1 2 0 1]
partition (1) index: [ 0  5  6 10], cluster labels: [0 0 0 0]
partition (2) index: [ 3  7  8 11], cluster labels: [1 1 1 1]
partition (3) index: [1 2 4 9], cluster labels: [2 2 2 2]
train fold index: [0 3 5], cluster labels: [0 0 0]
train fold index: [ 6 10], cluster labels: [0 0]
train fold index: [2 7], cluster labels: [1 1]
train fold index: [ 9 11], cluster labels: [1 1]
train fold index: [1 4], cluster labels: [2 2]
train fold index: [8], cluster labels: [2]
```

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

Method for storing array data.

Parameters

- **X** (*array-like of shape [n_samples, n_features]*) – input array.
- **y** (*array-like of shape [n_samples,]*) – labels.
- **job** (*str, ['fit', 'predict'] (default='fit')*) – type of estimation job. If 'fit', the indexer will be fitted, which involves fitting the estimator. Otherwise, the indexer will not be fitted (since it is not used for prediction).

Returns indexer with stores sample size data.

Return type instance

partition (*X=None*, *y=None*, *as_array=False*)

Get partition indices for training full subset estimators.

Returns the index range for each partition of X.

Parameters

- **X** (*array-like of shape [n_samples, n_features] , optional*) – the set to partition. The training label array is also, accepted, as only the first dimension is used. If `X` is not passed at instantiation, the `fit` method must be called before `generate`, or `X` must be passed as an argument of `generate`.
- **y** (*array-like of shape [n_samples,], optional*) – the labels of the set to partition.
- **as_array** (*bool (default = False)*) – whether to return partition as an index array. Otherwise tuples of `(start, stop)` indices are returned.

class mlens.base.indexer.**FoldIndex**(*n_splits=2*, *X=None*, *raise_on_exception=True*)

Bases: *mlens.base.indexer.BaseIndex*

Indexer that generates the full size of `X`.

K-Fold iterator that generates fold index tuples.

FoldIndex creates a generator that returns a tuple of stop and start positions to be used for numpy array slicing [stop:start]. Note that slicing works well for the test set, but for the training set it is recommended to concatenate the index for training data that comes before the current test set with the index for the training data that comes after. This can easily be achieved with:

```
for train_tup, test_tup in self.generate():
    train_slice = numpy.hstack([numpy.arange(t0, t1) for t0, t1 in
                                train_tup])

    xtrain, xtest = X[train_slice], X[test_tup[0]:test_tup[1]]
```

Warning: Simple clicing (i.e. `X[start:stop]` generally does not work for the train set, which often requires concatenating the train index range below the current test set, and the train index range above the current test set. To build get a training index, use

```
``hstack([np.arange(t0, t1) for t0, t1 in train_index_tuples])``.
```

See also:

BlendIndex, *SubsetIndex*

Examples

Creating arrays of folds and checking overlap

```
>>> import numpy as np
>>> from mlens.base.indexer import FoldIndex
>>> X = np.arange(10)
>>> print("Data set: %r" % X)
>>> print()
>>>
>>> idx = FoldIndex(4, X)
>>>
>>> for train, test in idx.generate(as_array=True):
...     print('TRAIN IDX: %32r | TEST IDX: %16r' % (train, test))
>>>
>>> print()
>>>
```

```
>>> for train, test in idx.generate(as_array=True):
...     print('TRAIN SET: %32r | TEST SET: %16r' % (X[train], X[test]))
>>>
>>> for train_idx, test_idx in idx.generate(as_array=True):
...     assert not any([i in X[test_idx] for i in X[train_idx]])
>>>
>>> print()
>>>
>>> print("No overlap between train set and test set.")
Data set: array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

TRAIN IDX: array([3, 4, 5, 6, 7, 8, 9]) | TEST IDX: array([0, 1, 2]) TRAIN IDX: array([0, 1, 2, 6, 7, 8, 9]) | TEST IDX: array([3, 4, 5]) TRAIN IDX: array([0, 1, 2, 3, 4, 5, 8, 9]) | TEST IDX: array([6, 7]) TRAIN IDX: array([0, 1, 2, 3, 4, 5, 6, 7]) | TEST IDX: array([8, 9])

TRAIN SET: array([3, 4, 5, 6, 7, 8, 9]) | TEST SET: array([0, 1, 2]) TRAIN SET: array([0, 1, 2, 6, 7, 8, 9]) | TEST SET: array([3, 4, 5]) TRAIN SET: array([0, 1, 2, 3, 4, 5, 8, 9]) | TEST SET: array([6, 7]) TRAIN SET: array([0, 1, 2, 3, 4, 5, 6, 7]) | TEST SET: array([8, 9])

No overlap between train set and test set.

Passing `n_splits = 1` without raising exception.

```
>>> import numpy as np
>>> from mlens.base.indexer import FoldIndex
>>> X = np.arange(3)
>>> print("Data set: %r" % X)
>>> print()
>>>
>>> idx = FoldIndex(1, X, raise_on_exception=False)
>>>
>>> for train, test in idx.generate(as_array=True):
...     print('TRAIN IDX: %10r | TEST IDX: %10r' % (train, test))
/../mlens/base/indexer.py:167: UserWarning: 'n_splits' is 1, will return
full index as both training set and test set.
warnings.warn("'n_splits' is 1, will return full index as "
```

Data set: array([0, 1, 2]) TRAIN IDX: array([0, 1, 2]) | TEST IDX: array([0, 1, 2])

fit (*X*, *y=None*, *job=None*)
Method for storing array data.

Parameters

- **X** (*array-like of shape [n_samples, optional]*) – array to _collect dimension data from.
- **y** (*None*) – for compatibility
- **job** (*None*) – for compatibility

Returns indexer with stores sample size data.

Return type instance

class mlens.base.indexer.**FullIndex** (*X=None*)
Bases: *mlens.base.indexer.BaseIndex*

Vacuous indexer to be used with final layers.

FullIndex is a compatibility class to be used with meta layers. It stores the sample size to be predicted for use with the `ParallelProcessing` job manager, and yields a `None, None` index when *generate* is called.

However, it is preferable to build code that avoids call the `generate` method when the indexer is known to be an instance of FullIndex for transparency and maintainability.

fit (*X*, *y=None*, *job=None*)
Store dimensionality data about X.

class mlens.base.indexer.**SubsetIndex** (*n_partitions=2*, *n_splits=2*, *X=None*, *raise_on_exception=True*)
Bases: *mlens.base.indexer.BaseIndex*

Subsample index generator.

Generates cross-validation folds according used to create `J` partitions of the data and `v` folds on each partition according to as per[2]:

1. Split `X` into `J` partitions
2. For each partition:
   (a) For each fold `v`, create train index of all idx not in `v`
   (b) Concatenate all the fold `v` indices into a test index for fold `v` that spans all partitions

Setting `J = 1` is equivalent to the `FullIndexer`, which returns standard K-Fold train and test set indices.

See also:

FoldIndex, *BlendIndex*, `Subsemble`

References

Parameters

- **n_partitions** (*int, list (default = 2)*) – Number of partitions to split data in. If `n_partitions=1`, *SubsetIndex* reduces to standard K-Fold.
- **n_splits** (*int (default = 2)*) – Number of splits to create in each partition. `n_splits` can not be 1 if `n_partition > 1`. Note that if `n_splits = 1`, both the train and test set will index the full data.
- **X** (*array-like of shape [n_samples,] , optional*) – the training set to partition. The training label array is also, accepted, as only the first dimension is used. If `X` is not passed at instantiation, the `fit` method must be called before `generate`, or `X` must be passed as an argument of `generate`.
- **raise_on_exception** (*bool (default = True)*) – whether to warn on suspicious slices or raise an error.

Examples

```
>>> import numpy as np
>>> from mlens.base import SubsetIndex
>>> X = np.arange(10)
>>> idx = SubsetIndex(3, X=X)
>>>
>>> print('Expected partitions of X:')
>>> print('J = 1: {!r}'.format(X[0:4]))
```

[2] Sapp, S., van der Laan, M. J., & Canny, J. (2014). Subsemble: an ensemble method for combining subset-specific algorithm fits. Journal of Applied Statistics, 41(6), 1247-1259. http://doi.org/10.1080/02664763.2013.864263

```
>>> print('J = 2: {!r}'.format(X[4:7]))
>>> print('J = 3: {!r}'.format(X[7:10]))
>>> print('SubsetIndexer partitions:')
>>> for i, part in enumerate(idx.partition(as_array=True)):
...     print('J = {}: {!r}'.format(i + 1, part))
>>> print('SubsetIndexer folds on partitions:')
>>> for i, (tri, tei) in enumerate(idx.generate()):
...     fold = i % 2 + 1
...     part = i // 2 + 1
...     train = np.hstack([np.arange(t0, t1) for t0, t1 in tri])
...     test = np.hstack([np.arange(t0, t1) for t0, t1 in tei])
>>>     print("J = %i | f = %i | "
...           "train: %15r | test: %r" % (part, fold, train, test))
Expected partitions of X:
J = 1: array([0, 1, 2, 3])
J = 2: array([4, 5, 6])
J = 3: array([7, 8, 9])
SubsetIndexer partitions:
J = 1: array([0, 1, 2, 3])
J = 2: array([4, 5, 6])
J = 3: array([7, 8, 9])
SubsetIndexer folds on partitions:
J = 1 | f = 1 | train:   array([2, 3]) | test: array([0, 1, 4, 5, 7, 8])
J = 1 | f = 2 | train:   array([0, 1]) | test: array([2, 3, 6, 9])
J = 2 | f = 1 | train:      array([6]) | test: array([0, 1, 4, 5, 7, 8])
J = 2 | f = 2 | train:   array([4, 5]) | test: array([2, 3, 6, 9])
J = 3 | f = 1 | train:      array([9]) | test: array([0, 1, 4, 5, 7, 8])
J = 3 | f = 2 | train:   array([7, 8]) | test: array([2, 3, 6, 9])
```

fit (*X*, *y=None*, *job=None*)
Method for storing array data.

Parameters

- **X** (*array-like of shape [n_samples, optional]*) – array to _collect dimension data from.
- **y** (*None*) – for compatibility
- **job** (*None*) – for compatibility

Returns indexer with stores sample size data.

Return type instance

partition (*X=None*, *as_array=False*)
Get partition indices for training full subset estimators.

Returns the index range for each partition of X.

Parameters

- **X** (*array-like of shape [n_samples,] , optional*) – the training set to partition. The training label array is also, accepted, as only the first dimension is used. If `X` is not passed at instantiation, the `fit` method must be called before `generate`, or `X` must be passed as an argument of `generate`.
- **as_array** (*bool (default = False)*) – whether to return partition as an index array. Otherwise tuples of `(start, stop)` indices are returned.

Module contents

ML-ENSEMBLE

author Sebastian Flennerhag

copyright 2017

licence MIT

Base modules

class mlens.base.**IdTrain**(*size=10*)
Bases: mlens.externals.sklearn.base.BaseEstimator

Container to identify training set.

Samples a random subset from set passed to the *fit* method, to allow identification of the training set in a *transform* or *predict* method.

Parameters **size** (*int*) – size to sample. A random subset of size [size, size] will be stored in the instance.

fit(*X*)
Sample a training set.

Parameters **X** (*array-like*) – training set to sample observations from.

Returns **self** – fitted instance with stored sample.

Return type obj

is_train(*X*)
Check if an array is the training set.

Parameters **X** (*array-like*) – training set to sample observations from.

Returns **self** – fitted instance with stored sample.

Return type obj

class mlens.base.**BlendIndex**(*test_size=0.5*, *train_size=None*, *X=None*, *raise_on_exception=True*)
Bases: *mlens.base.indexer.BaseIndex*

Indexer that generates two non-overlapping subsets of X.

Iterator that generates one training fold and one test fold that are non-overlapping and that may or may not partition all of X depending on the user's specification.

BlendIndex creates a singleton generator (has on iteration) that yields two tuples of `(start, stop)` integers that can be used for numpy array slicing (i.e. `X[stop:start]`). If a full array index is desired this can easily be achieved with:

```
for train_tup, test_tup in self.generate():
    train_slice = numpy.hstack([numpy.arange(t0, t1) for t0, t1 in
                                train_tup])

    test_slice = numpy.hstack([numpy.arange(t0, t1) for t0, t1 in
                               test_tup])
```

Parameters

- **test_size** (*int or float (default = 0.5)*) – Size of the test set. If `float`, assumed to be proportion of full data set.

- **train_size** (*int or float, optional*) – Size of test set. If not specified (i.e. `train_size = None`, train_size is equal to `n_samples - test_size`. If `float`, assumed to be a proportion of full data set. If `train_size` + `test_size` amount to less than the observations in the full data set, a subset of specified size will be used.
- **X** (*array-like of shape [n_samples,] , optional*) – the training set to partition. The training label array is also, accepted, as only the first dimension is used. If `X` is not passed at instantiation, the `fit` method must be called before `generate`, or `X` must be passed as an argument of `generate`.
- **raise_on_exception** (*bool (default = True)*) – whether to warn on suspicious slices or raise an error.

See also:

FoldIndex, *SubsetIndex*

Examples

Selecting an absolute test size, with train size as the remainder

```
>>> import numpy as np
>>> from mlens.base.indexer import BlendIndex
>>> X = np.arange(8)
>>> idx = BlendIndex(3, rebase=True)
>>> print('Test size: 3')
>>> for tri, tei in idx.generate(X):
...     print('TEST (idx | array): (%i, %i) | %r ' % (tei[0], tei[1],
...                                                   X[tei[0]:tei[1]]))
...     print('TRAIN (idx | array): (%i, %i) | %r ' % (tri[0], tri[1],
...                                                    X[tri[0]:tri[1]]))
Test size: 3
TEST (idx | array): (5, 8) | array([5, 6, 7])
TRAIN (idx | array): (0, 5) | array([0, 1, 2, 3, 4])
```

Selecting a test and train size less than the total

```
>>> import numpy as np
>>> from mlens.base.indexer import BlendIndex
>>> X = np.arange(8)
>>> idx = BlendIndex(3, 4, X)
>>> print('Test size: 3')
>>> print('Train size: 4')
>>> for tri, tei in idx.generate(X):
...     print('TEST (idx | array): (%i, %i) | %r ' % (tei[0], tei[1],
...                                                   X[tei[0]:tei[1]]))
...     print('TRAIN (idx | array): (%i, %i) | %r ' % (tri[0], tri[1],
...                                                    X[tri[0]:tri[1]]))
Test size: 3
Train size: 4
TEST (idx | array): (4, 7) | array([4, 5, 6])
TRAIN (idx | array): (0, 4) | array([0, 1, 2, 3])
```

Selecting a percentage of observations as test and train set

```
>>> import numpy as np
>>> from mlens.base.indexer import BlendIndex
```

```
>>> X = np.arange(8)
>>> idx = BlendIndex(0.25, 0.45, X)
>>> print('Test size: 25% * 8 = 2')
>>> print('Train size: 45% * 8 < 4 -> 3')
>>> for tri, tei in idx.generate(X):
...     print('TEST (idx | array): (%i, %i) | %r ' % (tei[0], tei[1],
...                                                  X[tei[0]:tei[1]]))
...     print('TRAIN (idx | array): (%i, %i) | %r ' % (tri[0], tri[1],
...                                                   X[tri[0]:tri[1]]))
Test size: 25% * 8 = 2
Train size: 50% * 8 < 4 ->
TEST (idx | array): (3, 5) | array([[3, 4]])
TRAIN (idx | array): (0, 3) | array([[0, 1, 2]])
```

Rebasing the test set to be 0-indexed

```
>>> import numpy as np
>>> from mlens.base.indexer import BlendIndex
>>> X = np.arange(8)
>>> idx = BlendIndex(3, rebase=True)
>>> print('Test size: 3')
>>> for tri, tei in idx.generate(X):
...     print('TEST tuple: (%i, %i) | array: %r' % (tei[0], tei[1],
...                                                 np.arange(tei[0],
...                                                           tei[1])))
Test size: 3
TEST tuple: (0, 3) | array: array([0, 1, 2])
```

fit (*X*, *y=None*, *job=None*)
Method for storing array data.

Parameters

- **X** (*array-like of shape [n_samples, optional]*) – array to _collect dimension data from.
- **y** (*None*) – for compatibility
- **job** (*None*) – for compatibility

Returns indexer with stores sample size data.

Return type instance

class mlens.base.**FoldIndex** (*n_splits=2*, *X=None*, *raise_on_exception=True*)
Bases: *mlens.base.indexer.BaseIndex*

Indexer that generates the full size of X.

K-Fold iterator that generates fold index tuples.

FoldIndex creates a generator that returns a tuple of stop and start positions to be used for numpy array slicing [stop:start]. Note that slicing works well for the test set, but for the training set it is recommended to concatenate the index for training data that comes before the current test set with the index for the training data that comes after. This can easily be achieved with:

```
for train_tup, test_tup in self.generate():
    train_slice = numpy.hstack([numpy.arange(t0, t1) for t0, t1 in
                                train_tup])

    xtrain, xtest = X[train_slice], X[test_tup[0]:test_tup[1]]
```

Warning: Simple clicing (i.e. `X[start:stop]` generally does not work for the train set, which often requires concatenating the train index range below the current test set, and the train index range above the current test set. To build get a training index, use

```
``hstack([np.arange(t0, t1) for t0, t1 in train_index_tuples])``.
```

See also:

BlendIndex, *SubsetIndex*

Examples

Creating arrays of folds and checking overlap

```
>>> import numpy as np
>>> from mlens.base.indexer import FoldIndex
>>> X = np.arange(10)
>>> print("Data set: %r" % X)
>>> print()
>>>
>>> idx = FoldIndex(4, X)
>>>
>>> for train, test in idx.generate(as_array=True):
...     print('TRAIN IDX: %32r | TEST IDX: %16r' % (train, test))
>>>
>>> print()
>>>
>>> for train, test in idx.generate(as_array=True):
...     print('TRAIN SET: %32r | TEST SET: %16r' % (X[train], X[test]))
>>>
>>> for train_idx, test_idx in idx.generate(as_array=True):
...     assert not any([i in X[test_idx] for i in X[train_idx]])
>>>
>>> print()
>>>
>>> print("No overlap between train set and test set.")
Data set: array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

TRAIN IDX: array([3, 4, 5, 6, 7, 8, 9]) | TEST IDX: array([0, 1, 2]) TRAIN IDX: array([0, 1, 2, 6, 7, 8, 9]) | TEST IDX: array([3, 4, 5]) TRAIN IDX: array([0, 1, 2, 3, 4, 5, 8, 9]) | TEST IDX: array([6, 7]) TRAIN IDX: array([0, 1, 2, 3, 4, 5, 6, 7]) | TEST IDX: array([8, 9])

TRAIN SET: array([3, 4, 5, 6, 7, 8, 9]) | TEST SET: array([0, 1, 2]) TRAIN SET: array([0, 1, 2, 6, 7, 8, 9]) | TEST SET: array([3, 4, 5]) TRAIN SET: array([0, 1, 2, 3, 4, 5, 8, 9]) | TEST SET: array([6, 7]) TRAIN SET: array([0, 1, 2, 3, 4, 5, 6, 7]) | TEST SET: array([8, 9])

No overlap between train set and test set.

Passing `n_splits = 1` without raising exception.

```
>>> import numpy as np
>>> from mlens.base.indexer import FoldIndex
>>> X = np.arange(3)
>>> print("Data set: %r" % X)
>>> print()
>>>
```

```
>>> idx = FoldIndex(1, X, raise_on_exception=False)
>>>
>>> for train, test in idx.generate(as_array=True):
...     print('TRAIN IDX: %10r | TEST IDX: %10r' % (train, test))
/../mlens/base/indexer.py:167: UserWarning: 'n_splits' is 1, will return
full index as both training set and test set.
warnings.warn("'n_splits' is 1, will return full index as "
```

Data set: array([0, 1, 2]) TRAIN IDX: array([0, 1, 2]) | TEST IDX: array([0, 1, 2])

fit (*X*, *y=None*, *job=None*)
Method for storing array data.

Parameters

- **X** (*array-like of shape [n_samples, optional]*) – array to _collect dimension data from.
- **y** (*None*) – for compatibility
- **job** (*None*) – for compatibility

Returns indexer with stores sample size data.

Return type instance

class mlens.base.**SubsetIndex** (*n_partitions=2*, *n_splits=2*, *X=None*, *raise_on_exception=True*)
Bases: *mlens.base.indexer.BaseIndex*

Subsample index generator.

Generates cross-validation folds according used to create J partitions of the data and v folds on each partition according to as per[1]:

1. Split X into J partitions
2. For each partition:
   (a) For each fold v, create train index of all idx not in v
   (b) Concatenate all the fold v indices into a test index for fold v that spans all partitions

Setting J = 1 is equivalent to the FullIndexer, which returns standard K-Fold train and test set indices.

See also:

FoldIndex, *BlendIndex*, Subsemble

References

Parameters

- **n_partitions** (*int, list (default = 2)*) – Number of partitions to split data in. If n_partitions=1, *SubsetIndex* reduces to standard K-Fold.
- **n_splits** (*int (default = 2)*) – Number of splits to create in each partition. n_splits can not be 1 if n_partition > 1. Note that if n_splits = 1, both the train and test set will index the full data.

[1] Sapp, S., van der Laan, M. J., & Canny, J. (2014). Subsemble: an ensemble method for combining subset-specific algorithm fits. Journal of Applied Statistics, 41(6), 1247-1259. http://doi.org/10.1080/02664763.2013.864263

- **X** (*array-like of shape [n_samples,] , optional*) – the training set to partition. The training label array is also, accepted, as only the first dimension is used. If X is not passed at instantiation, the fit method must be called before generate, or X must be passed as an argument of generate.
- **raise_on_exception** (*bool (default = True)*) – whether to warn on suspicious slices or raise an error.

Examples

```
>>> import numpy as np
>>> from mlens.base import SubsetIndex
>>> X = np.arange(10)
>>> idx = SubsetIndex(3, X=X)
>>>
>>> print('Expected partitions of X:')
>>> print('J = 1: {!r}'.format(X[0:4]))
>>> print('J = 2: {!r}'.format(X[4:7]))
>>> print('J = 3: {!r}'.format(X[7:10]))
>>> print('SubsetIndexer partitions:')
>>> for i, part in enumerate(idx.partition(as_array=True)):
...     print('J = {}: {!r}'.format(i + 1, part))
>>> print('SubsetIndexer folds on partitions:')
>>> for i, (tri, tei) in enumerate(idx.generate()):
...     fold = i % 2 + 1
...     part = i // 2 + 1
...     train = np.hstack([np.arange(t0, t1) for t0, t1 in tri])
...     test = np.hstack([np.arange(t0, t1) for t0, t1 in tei])
>>>     print("J = %i | f = %i | "
...           "train: %15r | test: %r" % (part, fold, train, test))
Expected partitions of X:
J = 1: array([0, 1, 2, 3])
J = 2: array([4, 5, 6])
J = 3: array([7, 8, 9])
SubsetIndexer partitions:
J = 1: array([0, 1, 2, 3])
J = 2: array([4, 5, 6])
J = 3: array([7, 8, 9])
SubsetIndexer folds on partitions:
J = 1 | f = 1 | train:   array([2, 3]) | test: array([0, 1, 4, 5, 7, 8])
J = 1 | f = 2 | train:   array([0, 1]) | test: array([2, 3, 6, 9])
J = 2 | f = 1 | train:      array([6]) | test: array([0, 1, 4, 5, 7, 8])
J = 2 | f = 2 | train:   array([4, 5]) | test: array([2, 3, 6, 9])
J = 3 | f = 1 | train:      array([9]) | test: array([0, 1, 4, 5, 7, 8])
J = 3 | f = 2 | train:   array([7, 8]) | test: array([2, 3, 6, 9])
```

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

Method for storing array data.

Parameters

- **X** (*array-like of shape [n_samples, optional]*) – array to _collect dimension data from.
- **y** (*None*) – for compatibility
- **job** (*None*) – for compatibility

Returns indexer with stores sample size data.

Return type instance

partition(*X=None*, *as_array=False*)
Get partition indices for training full subset estimators.

Returns the index range for each partition of X.

Parameters

- **X** (*array-like of shape [n_samples,] , optional*) – the training set to partition. The training label array is also, accepted, as only the first dimension is used. If `X` is not passed at instantiation, the `fit` method must be called before `generate`, or `X` must be passed as an argument of `generate`.
- **as_array** (*bool (default = False)*) – whether to return partition as an index array. Otherwise tuples of `(start, stop)` indices are returned.

class mlens.base.**FullIndex**(*X=None*)
Bases: *mlens.base.indexer.BaseIndex*

Vacuous indexer to be used with final layers.

FullIndex is a compatibility class to be used with meta layers. It stores the sample size to be predicted for use with the `ParallelProcessing` job manager, and yields a `None, None` index when *generate* is called. However, it is preferable to build code that avoids call the `generate` method when the indexer is known to be an instance of FullIndex for transparency and maintainability.

fit(*X*, *y=None*, *job=None*)
Store dimensionality data about X.

class mlens.base.**ClusteredSubsetIndex**(*estimator*, *n_partitions=2*, *n_splits=2*, *X=None*, *y=None*, *fit_estimator=True*, *attr='predict'*, *partition_on='X'*, *raise_on_exception=True*)
Bases: *mlens.base.indexer.BaseIndex*

Clustered Subsample index generator.

Generates cross-validation folds according used to create `J` partitions of the data and `v` folds on each partition according to as per[2]:

1. Split `X` into `J` partitions
2. For each partition:
   (a) For each fold `v`, create train index of all idx not in `v`
   (b) Concatenate all the fold `v` indices into a test index for fold `v` that spans all partitions

Setting `J = 1` is equivalent to the `FullIndexer`, which returns standard K-Fold train and test set indices.

ClusteredSubsetIndex uses a user-provided estimator to partition the data, in contrast to the *SubsetIndex* generator, which partitions data into randomly into equal sizes.

See also:

FoldIndex, *BlendIndex*, *SubsetIndex*

References

Parameters

[2] Sapp, S., van der Laan, M. J., & Canny, J. (2014). Subsemble: an ensemble method for combining subset-specific algorithm fits. Journal of Applied Statistics, 41(6), 1247-1259. http://doi.org/10.1080/02664763.2013.864263

- **estimator** (*instance*) – Estimator to use for clustering.
- **n_partitions** (*int*) – Number of partitions the estimator will create.
- **n_splits** (*int (default = 2)*) – Number of folds to create in each partition. `n_splits` can not be 1 if `n_partition > 1`. Note that if `n_splits = 1`, both the train and test set will index the full data.
- **fit_estimator** (*bool (default = True)*) – whether to fit the estimator separately before generating labels.
- **attr** (*str (default = 'predict')*) – the attribute to use for generating cluster membership labels.
- **X** (*array-like of shape [n_samples,] , optional*) – the training set to partition. The training label array is also, accepted, as only the first dimension is used. If `X` is not passed at instantiation, the `fit` method must be called before `generate`, or `X` must be passed as an argument of `generate`.
- **raise_on_exception** (*bool (default = True)*) – whether to warn on suspicious slices or raise an error.

Examples

```
>>> import numpy as np
>>> from sklearn.cluster import KMeans
>>> from mlens.base.indexer import ClusteredSubsetIndex
>>>
>>> km = KMeans(3, random_state=0)
>>> X = np.arange(12).reshape(-1, 1); np.random.shuffle(X)
>>> print("Data: {}".format(X.ravel()))
>>>
>>> s = ClusteredSubsetIndex(km)
>>> s.fit(X)
>>>
>>> P = s.estimator.predict(X)
>>> print("cluster labels: {}".format(P))
>>>
>>> for j, i in enumerate(s.partition(as_array=True)):
...     print("partition ({}) index: {}, cluster labels: {}".format(i, j + 1,␣
↪P[i]))
>>>
>>> for i in s.generate(as_array=True):
...     print("train fold index: {}, cluster labels: {}".format(i[0], P[i[0]]))
Data: [ 8  7  5  2  4 10 11  1  3  6  9  0]
cluster labels: [0 2 2 1 2 0 0 1 1 2 0 1]
partition (1) index: [ 0  5  6 10], cluster labels: [0 0 0 0]
partition (2) index: [ 3  7  8 11], cluster labels: [1 1 1 1]
partition (3) index: [1 2 4 9], cluster labels: [2 2 2 2]
train fold index: [0 3 5], cluster labels: [0 0 0]
train fold index: [ 6 10], cluster labels: [0 0]
train fold index: [2 7], cluster labels: [1 1]
train fold index: [ 9 11], cluster labels: [1 1]
train fold index: [1 4], cluster labels: [2 2]
train fold index: [8], cluster labels: [2]
```

fit (*X*, *y=None*, *job='fit'*)
Method for storing array data.

Parameters

- **X** (*array-like of shape [n_samples, n_features]*) – input array.
- **y** (*array-like of shape [n_samples,]*) – labels.
- **job** (*str, ['fit', 'predict'] (default='fit')*) – type of estimation job. If 'fit', the indexer will be fitted, which involves fitting the estimator. Otherwise, the indexer will not be fitted (since it is not used for prediction).

Returns indexer with stores sample size data.

Return type instance

partition (*X=None*, *y=None*, *as_array=False*)
Get partition indices for training full subset estimators.

Returns the index range for each partition of X.

Parameters

- **X** (*array-like of shape [n_samples, n_features] , optional*) – the set to partition. The training label array is also, accepted, as only the first dimension is used. If `X` is not passed at instantiation, the `fit` method must be called before `generate`, or `X` must be passed as an argument of `generate`.
- **y** (*array-like of shape [n_samples,], optional*) – the labels of the set to partition.
- **as_array** (*bool (default = False)*) – whether to return partition as an index array. Otherwise tuples of `(start, stop)` indices are returned.

mlens.ensemble package

Submodules

mlens.ensemble.base module

ML-ENSEMBLE

author Sebastian Flennerhag

copyright 2017

licence MIT

Base classes for ensemble layer management.

class `mlens.ensemble.base.`**BaseEnsemble** (*shuffle=False*, *random_state=None*, *scorer=None*, *raise_on_exception=True*, *verbose=False*, *n_jobs=-1*, *layers=None*, *array_check=2*, *backend=None*)
Bases: `mlens.externals.sklearn.base.BaseEstimator`

BaseEnsemble class.

Core ensemble class methods used to add ensemble layers and manipulate parameters.

fit (*X*, *y=None*)
Fit ensemble.

Parameters

- **X** (*array-like of shape = [n_samples, n_features]*) – input matrix to be used for prediction.
- **y** (*array-like of shape = [n_samples,] or None (default = None)*) – output vector to trained estimators on.

Returns **self** – class instance with fitted estimators.

Return type instance

predict(*X*)

Predict with fitted ensemble.

Parameters **X** (*array-like, shape=[n_samples, n_features]*) – input matrix to be used for prediction.

Returns **y_pred** – predictions for provided input array.

Return type array-like, shape=[n_samples,]

predict_proba(*X*)

Predict class probabilities with fitted ensemble.

Compatibility method for Scikit-learn. This method checks that the final layer has `proba=True`, then calls the regular `predict` method.

Parameters **X** (*array-like, shape=[n_samples, n_features]*) – input matrix to be used for prediction.

Returns **y_pred** – predicted class membership probabilities for provided input array.

Return type array-like, shape=[n_samples, n_classes]

set_verbosity(*verbose*)

Adjust the level of verbosity.

class mlens.ensemble.base.**Layer**(*estimators*, *cls*, *indexer=None*, *preprocessing=None*, *proba=False*, *partitions=1*, *propagate_features=None*, *scorer=None*, *raise_on_exception=False*, *name=None*, *dtype=None*, *verbose=False*, *cls_kwargs=None*)

Bases: mlens.externals.sklearn.base.BaseEstimator

Layer of preprocessing pipes and estimators.

Layer is an internal class that holds a layer and its associated data including an estimation procedure. It behaves as an estimator from an Scikit-learn API point of view.

Parameters

- **estimators** (*dict of lists or list*) – estimators constituting the layer. If `preprocessing` is `None` or `list`, `estimators` should be a `list`. The list can either contain estimator instances, named tuples of estimator instances, or a combination of both.

```
option_1 = [estimator_1, estimator_2]
option_2 = [("est-1", estimator_1), ("est-2", estimator_2)]
option_3 = [estimator_1, ("est-2", estimator_2)]
```

If different preprocessing pipelines are desired, a dictionary that maps estimators to preprocessing pipelines must be passed. The names of the estimator dictionary must correspond to the names of the estimator dictionary.

```
preprocessing_cases = {"case-1": [trans_1, trans_2].
                       "case-2": [alt_trans_1, alt_trans_2]}
```

```
estimators = {"case-1": [est_a, est_b].
              "case-2": [est_c, est_d]}
```

The lists for each dictionary entry can be any of `option_1`, `option_2` and `option_3`.

- **cls** (*str*) – type of layers. Should be the name of an accepted estimator class.
- **indexer** (*instance, optional*) – Indexer instance to use. Defaults to the layer class indexer instantiated with default settings. Required arguments depend on the indexer. See *mlens.base* for details.
- **preprocessing** (*dict of lists or list, optional (default = None)*) – preprocessing pipelines for given layer. If the same preprocessing applies to all estimators, `preprocessing` should be a list of transformer instances. The list can contain the instances directly, named tuples of transformers, or a combination of both.

```
option_1 = [transformer_1, transformer_2]
option_2 = [("trans-1", transformer_1),
            ("trans-2", transformer_2)]
option_3 = [transformer_1, ("trans-2", transformer_2)]
```

If different preprocessing pipelines are desired, a dictionary that maps preprocessing pipelines must be passed. The names of the preprocessing dictionary must correspond to the names of the estimator dictionary.

```
preprocessing_cases = {"case-1": [trans_1, trans_2].
                       "case-2": [alt_trans_1, alt_trans_2]}

estimators = {"case-1": [est_a, est_b].
              "case-2": [est_c, est_d]}
```

The lists for each dictionary entry can be any of `option_1`, `option_2` and `option_3`.

- **proba** (*bool (default = False)*) – whether to call *predict_proba* on the estimators in the layer when predicting.
- **partitions** (*int (default = 1)*) – Number of subset-specific fits to generate from the learner library.
- **propagate_features** (*list, optional*) – Features to propagate from the input array to the output array. Carries input features to the output of the layer, useful for propagating original data through several stacked layers. Propagated features are stored in the left-most columns.
- **raise_on_exception** (*bool (default = False)*) – whether to raise an error on soft exceptions, else issue warning.
- **verbose** (*int or bool (default = False)*) – level of verbosity.
  - `verbose = 0` silent (same as `verbose = False`)
  - `verbose = 1` messages at start and finish (same as `verbose = True`)
  - `verbose = 2` messages for each layer

  If `verbose >= 50` prints to `sys.stdout`, else `sys.stderr`. For verbosity in the layers themselves, use `fit_params`.
- **dtype** (numpy dtype class, default = `numpy.float32`) – dtype format of prediction array.
- **cls_kwargs** (*dict or None*) – optional arguments to pass to the layer type class.

estimators_
OrderedDict, list – container for fitted estimators, possibly mapped to preprocessing cases and / or folds.

preprocessing_
OrderedDict, list – container for fitted preprocessing pipelines, possibly mapped to preprocessing cases and / or folds.

get_params(*deep=True*)
Get parameters for this estimator.

Parameters **deep** (*boolean (default = True)*) – If `True`, will return the layers separately as individual parameters. If `False`, will return the collapsed dictionary.

Returns **params** – mapping of parameter names mapped to their values.

Return type dict

class `mlens.ensemble.base.`**LayerContainer**(*layers=None*, *n_jobs=-1*, *backend=None*, *raise_on_exception=False*, *verbose=False*)
Bases: `mlens.externals.sklearn.base.BaseEstimator`

Container class for layers.

The LayerContainer class stories all layers as an ordered dictionary and modifies possesses a `get_params` method to appear as an estimator in the Scikit-learn API. This allows correct cloning and parameter updating.

Parameters

- **layers** (*OrderedDict, None (default = None)*) – An ordered dictionary of Layer instances. To initiate a new `LayerContainer` instance, set `layers = None`.
- **n_jobs** (*int (default = -1)*) – Number of CPUs to use. Set `n_jobs = -1` for all available CPUs, and `n_jobs = -2` for all available CPUs except one, e.tc..
- **backend** (*str, (default="threading")*) – the joblib backend to use (i.e. "multiprocessing" or "threading").
- **raise_on_exception** (*bool (default = False)*) – raise error on soft exceptions. Otherwise issue warning.
- **verbose** (*int or bool (default = False)*) – level of verbosity.
  - `verbose = 0` silent (same as `verbose = False`)
  - `verbose = 1` messages at start and finish (same as `verbose = True`)
  - `verbose = 2` messages for each layer

  If `verbose >= 50` prints to `sys.stdout`, else `sys.stderr`. For verbosity in the layers themselves, use `fit_params`.

add(*estimators*, *cls*, *indexer=None*, *preprocessing=None*, ***kwargs*)
Method for adding a layer.

Parameters

- **estimators** (*dict of lists or list*) – estimators constituting the layer. If `preprocessing` is `None` or `list`, `estimators` should be a `list`. The list can either contain estimator instances, named tuples of estimator instances, or a combination of both.

```
option_1 = [estimator_1, estimator_2]
option_2 = [("est-1", estimator_1), ("est-2", estimator_2)]
option_3 = [estimator_1, ("est-2", estimator_2)]
```

If different preprocessing pipelines are desired, a dictionary that maps estimators to preprocessing pipelines must be passed. The names of the estimator dictionary must correspond to the names of the estimator dictionary.

```
preprocessing_cases = {"case-1": [trans_1, trans_2].
                       "case-2": [alt_trans_1, alt_trans_2]}

estimators = {"case-1": [est_a, est_b].
              "case-2": [est_c, est_d]}
```

The lists for each dictionary entry can be any of `option_1`, `option_2` and `option_3`.

- **`cls`** (*`str`*) – Type of layer, as defined by the estimation class to instantiate when processing a layer. See *`mlens.ensemble`* for available classes.
- **`indexer`** (*`instance or None (default = None)`*) – Indexer instance to use. Defaults to the layer class indexer with default settings. See *`mlens.base`* for details.
- **`preprocessing`** (*`dict of lists or list, optional (default = None)`*) – preprocessing pipelines for given layer. If the same preprocessing applies to all estimators, `preprocessing` should be a list of transformer instances. The list can contain the instances directly, named tuples of transformers, or a combination of both.

```
option_1 = [transformer_1, transformer_2]
option_2 = [("trans-1", transformer_1),
            ("trans-2", transformer_2)]
option_3 = [transformer_1, ("trans-2", transformer_2)]
```

If different preprocessing pipelines are desired, a dictionary that maps preprocessing pipelines must be passed. The names of the preprocessing dictionary must correspond to the names of the estimator dictionary.

```
preprocessing_cases = {"case-1": [trans_1, trans_2].
                       "case-2": [alt_trans_1, alt_trans_2]}

estimators = {"case-1": [est_a, est_b].
              "case-2": [est_c, est_d]}
```

The lists for each dictionary entry can be any of `option_1`, `option_2` and `option_3`.

- **`**kwargs`** (*`optional`*) – keyword arguments to be passed onto the layer at instantiation.

Returns **self** – if `in_place = True`, returns `self` with the layer instantiated.

Return type instance, optional

`fit` (*X=None*, *y=None*, *return_preds=None*, ***process_kwargs*)
Fit instance by calling `predict_proba` in the first layer.

Similar to `fit`, but will call the `predict_proba` method on estimators. Thus, each the `n_test_samples * n_labels` prediction matrix of each estimator will be stacked and used as input in the subsequent layer.

Parameters

- **`X`** (*`array-like of shape = [n_samples, n_features]`*) – input matrix to be used for fitting and predicting.
- **`y`** (*`array-like of shape = [n_samples, ]`*) – training labels.
- **`return_preds`** (*`bool`*) – whether to return final prediction array

- ****process_kwargs** (*optional*) – optional arguments to initialize processor with.

Returns

- **out** (*dict*) – dictionary of output data (possibly empty) generated through fitting. Keys correspond to layer names and values to the output generated by calling the layer's `fit_function`.

```
out = {'layer-i-estimator-j': some_data,
       ...
       'layer-s-estimator-q': some_data}
```

- **X** (*array-like, optional*) – predictions from final layer's `fit_proba` call.

get_params(*deep=True*)

Get parameters for this estimator.

Parameters **deep** (*boolean, optional*) – If True, will return the layers separately as individual parameters. If False, will return the collapsed dictionary.

Returns **params** – mapping of parameter names mapped to their values.

Return type dict

predict(*X=None*, **args*, ***kwargs*)

Generic method for predicting through all layers in the container.

Parameters

- **X** (*array-like of shape = [n_samples, n_features]*) – input matrix to be used for prediction.
- ***args** (*optional*) – optional arguments.
- ****kwargs** (*optional*) – optional keyword arguments.

Returns **X_pred** – predictions from final layer.

Return type array-like of shape = [n_samples, n_fitted_estimators]

transform(*X=None*, **args*, ***kwargs*)

Generic method for reproducing predictions of the `fit` call.

Parameters

- **X** (*array-like of shape = [n_samples, n_features]*) – input matrix to be used for prediction.
- ***args** (*optional*) – optional arguments.
- ****kwargs** (*optional*) – optional keyword arguments.

Returns **X_pred** – predictions from `fit` call to final layer.

Return type array-like of shape = [n_test_samples, n_fitted_estimators]

mlens.ensemble.base.**print_job**(*lc*, *start_message*)

Print job details.

Parameters

- **lc** (*LayerContainer*) – The LayerContainer instance running the job.
- **start_message** (*str*) – Initial message.

mlens.ensemble.blend module

ML-ENSEMBLE

author Sebastian Flennerhag

copyright 2017

licence MIT

Blend Ensemble class. Fully integrable with Scikit-learn.

class mlens.ensemble.blend.**BlendEnsemble**(*test_size=0.5*, *shuffle=False*, *random_state=None*, *scorer=None*, *raise_on_exception=True*, *array_check=2*, *verbose=False*, *n_jobs=-1*, *backend=None*, *layers=None*)

Bases: *mlens.ensemble.base.BaseEnsemble*

Blend Ensemble class.

The Blend Ensemble is a supervised ensemble closely related to the `SuperLearner`. It differs in that to estimate the prediction matrix Z used by the meta learner, it uses a subset of the data to predict its complement, and the meta learner is fitted on those predictions.

By only fitting every base learner once on a subset of the full training data, *BlendEnsemble* is a fast ensemble that can handle very large datasets simply by only using portion of it at each stage. The cost of this approach is that information is thrown out at each stage, as one layer will not see the training data used by the previous layer.

With large data that can be expected to satisfy an i.i.d. assumption, the *BlendEnsemble* can achieve similar performance to more sophisticated ensembles at a fraction of the training time. However, with data data is not uniformly distributed or exhibits high variance the *BlendEnsemble* can be a poor choice as information is lost at each stage of fitting.

See also:

`SuperLearner`, `Subsemble`

Parameters

- **test_size** (*int, float (default = 0.5)*) – the size of the test set for each layer. This parameter can be overridden in the `add` method if different test sizes is desired for each layer. If a `float` is specified, it is presumed to be the fraction of the available data to be used for training, and so `0. < test_size < 1.`.
- **shuffle** (*bool (default = True)*) – whether to shuffle data before selecting training data.
- **random_state** (*int (default = None)*) – random seed if shuffling inputs.
- **scorer** (*object (default = None)*) – scoring function. If a function is provided, base estimators will be scored on the prediction made. The scorer should be a function that accepts an array of true values and an array of predictions: `score = f(y_true, y_pred)`.
- **raise_on_exception** (*bool (default = True)*) – whether to issue warnings on soft exceptions or raise error. Examples include lack of layers, bad inputs, and failed fit of an estimator in a layer. If set to `False`, warnings are issued instead but estimation continues unless exception is fatal. Note that this can result in unexpected behavior unless the exception is anticipated.
- **array_check** (*int (default = 2)*) – level of strictness in checking input arrays.

- `array_check = 0` will not check `X` or `y`
- `array_check = 1` will check `X` and `y` for inconsistencies and warn when format looks suspicious, but retain original format.
- `array_check = 2` will impose Scikit-learn array checks, which converts `X` and `y` to numpy arrays and raises an error if conversion fails.

- **verbose** (*int or bool (default = False)*) – level of verbosity.
  - `verbose = 0` silent (same as `verbose = False`)
  - `verbose = 1` messages at start and finish (same as `verbose = True`)
  - `verbose = 2` messages for each layer

  If `verbose >= 50` prints to `sys.stdout`, else `sys.stderr`. For verbosity in the layers themselves, use `fit_params`.
- **n_jobs** (*int (default = -1)*) – number of CPU cores to use for fitting and prediction.
- **backend** (*str or object (default = 'threading')*) – backend infrastructure to use during call to `mlens.externals.joblib.Parallel`. See Joblib for further documentation. To set global backend, set `mlens.config.BACKEND`.

scores_
dict – if `scorer` was passed to instance, `scores_` contains dictionary with cross-validated scores assembled during `fit` call. The fold structure used for scoring is determined by `folds`.

Examples

Instantiate ensembles with no preprocessing: use list of estimators

```
>>> from mlens.ensemble import BlendEnsemble
>>> from mlens.metrics.metrics import rmse
>>> from sklearn.datasets import load_boston
>>> from sklearn.linear_model import Lasso
>>> from sklearn.svm import SVR
>>>
>>> X, y = load_boston(True)
>>>
>>> ensemble = BlendEnsemble()
>>> ensemble.add([SVR(), ('can name some or all est', Lasso())])
>>> ensemble.add_meta(SVR())
>>>
>>> ensemble.fit(X, y)
>>> preds = ensemble.predict(X)
>>> rmse(y, preds)
7.656098...
```

Instantiate ensembles with different preprocessing pipelines through dicts.

```
>>> from mlens.ensemble import BlendEnsemble
>>> from mlens.metrics.metrics import rmse
>>> from sklearn.datasets import load_boston
>>> from sklearn. preprocessing import MinMaxScaler, StandardScaler
>>> from sklearn.linear_model import Lasso
>>> from sklearn.svm import SVR
```

```
>>>
>>> X, y = load_boston(True)
>>>
>>> preprocessing_cases = {'mm': [MinMaxScaler()],
...                        'sc': [StandardScaler()]}
>>>
>>> estimators_per_case = {'mm': [SVR()],
...                        'sc': [('can name some or all ests', Lasso())]}
>>>
>>> ensemble = BlendEnsemble()
>>> ensemble.add(estimators_per_case, preprocessing_cases).add(SVR(),
...                                                            meta=True)
>>>
>>> ensemble.fit(X, y)
>>> preds = ensemble.predict(X)
>>> rmse(y, preds)
7.9814242...
```

add(*estimators*, *preprocessing=None*, *test_size=None*, *proba=False*, *meta=False*, *propagate_features=None*, ***kwargs*)
Add layer to ensemble.

Parameters

- **preprocessing** (*dict of lists or list, optional (default = None)*) – preprocessing pipelines for given layer. If the same preprocessing applies to all estimators, `preprocessing` should be a list of transformer instances. The list can contain the instances directly, named tuples of transformers, or a combination of both.

```
option_1 = [transformer_1, transformer_2]
option_2 = [("trans-1", transformer_1),
            ("trans-2", transformer_2)]
option_3 = [transformer_1, ("trans-2", transformer_2)]
```

If different preprocessing pipelines are desired, a dictionary that maps preprocessing pipelines must be passed. The names of the preprocessing dictionary must correspond to the names of the estimator dictionary.

```
preprocessing_cases = {"case-1": [trans_1, trans_2],
                       "case-2": [alt_trans_1, alt_trans_2]}

estimators = {"case-1": [est_a, est_b],
              "case-2": [est_c, est_d]}
```

The lists for each dictionary entry can be any of `option_1`, `option_2` and `option_3`.

- **estimators** (*dict of lists or list or instance*) – estimators constituting the layer. If preprocessing is none and the layer is meant to be the meta estimator, it is permissible to pass a single instantiated estimator. If `preprocessing` is `None` or `list`, `estimators` should be a `list`. The list can either contain estimator instances, named tuples of estimator instances, or a combination of both.

```
option_1 = [estimator_1, estimator_2]
option_2 = [("est-1", estimator_1), ("est-2", estimator_2)]
option_3 = [estimator_1, ("est-2", estimator_2)]
```

If different preprocessing pipelines are desired, a dictionary that maps estimators to preprocessing pipelines must be passed. The names of the estimator dictionary must corre-

spond to the names of the estimator dictionary.

```
preprocessing_cases = {"case-1": [trans_1, trans_2],
                       "case-2": [alt_trans_1, alt_trans_2]}

estimators = {"case-1": [est_a, est_b],
              "case-2": [est_c, est_d]}
```

The lists for each dictionary entry can be any of `option_1`, `option_2` and `option_3`.

- **test_size** (*int or float, optional*) – Use if a different test set size is desired for layer than what the ensemble was instantiated with.
- **proba** (*bool (default = False)*) – Whether to call `predict_proba` on base learners.
- **propagate_features** (*list, optional*) – List of column indexes to propagate from the input of the layer to the output of the layer. Propagated features are concatenated and stored in the leftmost columns of the output matrix. The `propagate_features` list should define a slice of the numpy array containing the input data, e.g. `[0, 1]` to propagate the first two columns of the input matrix to the output matrix.
- **meta** (*bool (default = False)*) – Whether the layer should be treated as the final meta estimator.
- ****kwargs** (*optional*) – optional keyword arguments to instantiate layer with.

Returns **self** – ensemble instance with layer instantiated.

Return type instance

add_meta(*estimator*, ***kwargs*)

Meta Learner.

Compatibility method for adding a meta learner to be used for final predictions.

Parameters

- **estimator** (*instance*) – estimator instance.
- ****kwargs** (*optional*) – optional keyword arguments.

mlens.ensemble.sequential module

ML-ENSEMBLE

author Sebastian Flennerhag

copyright 2017

licence MIT

Sequential Ensemble class. Fully integrable with Scikit-learn.

class mlens.ensemble.sequential.**SequentialEnsemble**(*shuffle=False*, *random_state=None*, *scorer=None*, *raise_on_exception=True*, *array_check=2*, *verbose=False*, *n_jobs=-1*, *backend=None*, *layers=None*)

Bases: *mlens.ensemble.base.BaseEnsemble*

Sequential Ensemble class.

The Sequential Ensemble class allows users to build ensembles with different classes of layers. The type of layer and its parameters are specified when added to the ensemble. See respective ensemble class for details on parameters.

See also:

`BlendEnsemble`, `Subsemble`, `SuperLearner`

Parameters

- **shuffle** (*bool (default = True)*) – whether to shuffle data before generating folds.
- **random_state** (*int (default = None)*) – random seed if shuffling inputs.
- **scorer** (*object (default = None)*) – scoring function. If a function is provided, base estimators will be scored on the training set assembled for fitting the meta estimator. Since those predictions are out-of-sample, the scores represent valid test scores. The scorer should be a function that accepts an array of true values and an array of predictions: `score = f(y_true, y_pred)`.
- **raise_on_exception** (*bool (default = True)*) – whether to issue warnings on soft exceptions or raise error. Examples include lack of layers, bad inputs, and failed fit of an estimator in a layer. If set to `False`, warnings are issued instead but estimation continues unless exception is fatal. Note that this can result in unexpected behavior unless the exception is anticipated.
- **array_check** (*int (default = 2)*) – level of strictness in checking input arrays.
  - `array_check = 0` will not check `X` or `y`
  - `array_check = 1` will check `X` and `y` for inconsistencies and warn when format looks suspicious, but retain original format.
  - `array_check = 2` will impose Scikit-learn array checks, which converts `X` and `y` to numpy arrays and raises an error if conversion fails.
- **verbose** (*int or bool (default = False)*) – level of verbosity.
  - `verbose = 0` silent (same as `verbose = False`)
  - `verbose = 1` messages at start and finish (same as `verbose = True`)
  - `verbose = 2` messages for each layer

  If `verbose >= 50` prints to `sys.stdout`, else `sys.stderr`. For verbosity in the layers themselves, use `fit_params`.
- **n_jobs** (*int (default = -1)*) – number of CPU cores to use for fitting and prediction.
- **backend** (*str or object (default = 'threading')*) – backend infrastructure to use during call to `mlens.externals.joblib.Parallel`. See Joblib for further documentation. To change global backend, set `mlens.config.BACKEND`

scores_

dict – if `scorer` was passed to instance, `scores_` contains dictionary with cross-validated scores assembled during `fit` call. The fold structure used for scoring is determined by `folds`.

Examples

```
>>> from mlens.ensemble import SequentialEnsemble
>>> from mlens.metrics.metrics import rmse
>>> from sklearn.datasets import load_boston
>>> from sklearn.linear_model import Lasso
>>> from sklearn.svm import SVR
>>>
>>> X, y = load_boston(True)
>>>
>>> ensemble = SequentialEnsemble()
>>>
>>> # Add a subsemble with 5 partitions as first layer
>>> ensemble.add('subset', [SVR(), Lasso()], n_partitions=10, n_splits=10)
>>>
>>> # Add a super learner as second layer
>>> ensemble.add('stack', [SVR(), Lasso()], n_splits=20)
>>>
>>> # Specify a meta estimator
>>> ensemble.add_meta(SVR())
>>>
>>> ensemble.fit(X, y)
>>> preds = ensemble.predict(X)
>>> rmse(y, preds)
6.5628...
```

add(*cls*, *estimators*, *preprocessing=None*, ***kwargs*)

Add layer to ensemble.

For full set of optional arguments, see the ensemble API for the specified type.

Parameters

- **cls** (*str*) – layer class. Accepted types are:
  - 'blend' : blend ensemble
  - 'subset' : subsemble
  - 'stack' : super learner
- **estimators** (*dict of lists or list or instance*) – estimators constituting the layer. If preprocessing is none and the layer is meant to be the meta estimator, it is permissible to pass a single instantiated estimator. If `preprocessing` is `None` or `list`, `estimators` should be a `list`. The list can either contain estimator instances, named tuples of estimator instances, or a combination of both.

```
option_1 = [estimator_1, estimator_2]
option_2 = [("est-1", estimator_1), ("est-2", estimator_2)]
option_3 = [estimator_1, ("est-2", estimator_2)]
```

If different preprocessing pipelines are desired, a dictionary that maps estimators to preprocessing pipelines must be passed. The names of the estimator dictionary must correspond to the names of the estimator dictionary.

```
preprocessing_cases = {"case-1": [trans_1, trans_2],
                       "case-2": [alt_trans_1, alt_trans_2]}

estimators = {"case-1": [est_a, est_b],
              "case-2": [est_c, est_d]}
```

The lists for each dictionary entry can be any of `option_1`, `option_2` and `option_3`.

- **preprocessing** (*dict of lists or list, optional (default = None)*) – preprocessing pipelines for given layer. If the same preprocessing applies to all estimators, `preprocessing` should be a list of transformer instances. The list can contain the instances directly, named tuples of transformers, or a combination of both.

```
option_1 = [transformer_1, transformer_2]
option_2 = [("trans-1", transformer_1),
            ("trans-2", transformer_2)]
option_3 = [transformer_1, ("trans-2", transformer_2)]
```

If different preprocessing pipelines are desired, a dictionary that maps preprocessing pipelines must be passed. The names of the preprocessing dictionary must correspond to the names of the estimator dictionary.

```
preprocessing_cases = {"case-1": [trans_1, trans_2],
                       "case-2": [alt_trans_1, alt_trans_2]}

estimators = {"case-1": [est_a, est_b],
              "case-2": [est_c, est_d]}
```

The lists for each dictionary entry can be any of `option_1`, `option_2` and `option_3`.

- ****kwargs** (*optional*) – optional keyword arguments to instantiate layer with. See respective ensemble for further details.

Returns **self** – ensemble instance with layer instantiated.

Return type instance

add_meta(*estimator*, ***kwargs*)
Meta Learner.

Meta learner to be used for final predictions.

Parameters

- **estimator** (*instance*) – estimator instance.
- ****kwargs** (*optional*) – optional keyword arguments.

mlens.ensemble.subsemble module

ML-ENSEMBLE

author Sebastian Flennerhag

copyright 2017

licence MIT

Subsemble class. Fully integrable with Scikit-learn.

class `mlens.ensemble.subsemble.`**Subsemble**(*partitions=2*, *partition_estimator=None*, *folds=2*, *shuffle=False*, *random_state=None*, *scorer=None*, *raise_on_exception=True*, *array_check=2*, *verbose=False*, *n_jobs=-1*, *backend=None*, *layers=None*)
Bases: *mlens.ensemble.base.BaseEnsemble*

Subsemble class.

Subsemble is a supervised ensemble algorithm that uses subsets of the full data to fit a layer, and within each subset K-fold estimation to map a training set (X, y) into a prediction set (Z, y), where Z is a matrix of prediction from each estimator on each subset (thus of shape `[n_samples, (n_partitions * n_estimators)]`). Z is constructed using K-Fold splits of each partition of *X* to ensure Z reflects test errors within each partition. A final user-specified meta learner is fitted to the final ensemble layer's prediction, to learn the best combination of subset-specific estimator predictions. By passing a `partition_estimator`, the partitions can be learnt. The algorithm in sudo code :

1. For each layer in the ensemble, do:
   (a) Specify a library of L base learners
   (b) Specify a partition strategy and partition X into J subsets.
   (c) For each partition do:
      i. Fit all base learners and store them
      ii. Create K folds
      iii. For each fold, do:
         A. Fit all base learners on the training folds
         B. Collect *all* test folds, across partitions, and predict.
   (d) Assemble a cross-validated prediction matrix $Z \in \mathbb{R}^{(n \times (L \times J))}$ by stacking predictions made in the cross-validation step.
2. Fit the meta learner on Z and store the learner.

The ensemble can be used for prediction by mapping a new test set T into a prediction set Z' using the learners fitted in (1.3.1), and then using Z' to generate final predictions through the fitted meta learner from (2).

The Subsemble does asymptotically as well as (up to a constant) the Oracle selector. For the theory behind the Subsemble, see[1] and references therein.

By partitioning the data into subset and fitting on those, a Subsemble can reduce training time considerably if estimators does not scale linearly. Moreover, Subsemble allows estimators to learn different patterns from each subset, and so can improve the overall performance by achieving a tighter fit on each subset. Since all observations in the training set are predicted, no information is lost between layers.

This implementation allows very general partition estimators. The user must ensure that the partition estimator behaves as desired. To alter the expected behavior, see the `kwd` parameter under the `add` method and the *`mlens.base.ClusteredSubsetIndex`*. Also see the advanced tutorials for example use cases.

References

See also:

`BlendEnsemble`, `SuperLearner`

Parameters

- **`partitions`** (*`int (default = 2)`*) – number of partitions to split data into. For each layer, increasing partitions increases the number of estimators in the ensemble by a factor equal to the number of estimators. Note: this parameter can be specified on a layer-specific basis in the *`add`* method.

[1] Sapp, S., van der Laan, M. J., & Canny, J. (2014). Subsemble: an ensemble method for combining subset-specific algorithm fits. Journal of Applied Statistics, 41(6), 1247-1259. http://doi.org/10.1080/02664763.2013.864263

- **partition_estimator** (*instance, optional*) – To use a supervised or unsupervised estimator to learn partitions, pass an instantiated estimator as `partition_estimator`. The estimator must accept a `fit` call for fitting the training data, and a `predict` call that *assigns cluster partitions labels*. For instance, clustering estimator or classifiers (where their class predictions will be used for partitioning). The number of partitions by the estimator must correspond to the `partitions` argument. Specific estimators can be added to each layer by passing the estimator during the call to the ensemble's `add` method.
- **folds** (*int (default = 2)*) – number of folds to use during fitting. Note: this parameter can be specified on a layer-specific basis in the *add* method.
- **shuffle** (*bool (default = True)*) – whether to shuffle data before generating folds.
- **random_state** (*int (default = None)*) – random seed if shuffling inputs.
- **scorer** (*object (default = None)*) – scoring function. If a function is provided, base estimators will be scored on the training set assembled for fitting the meta estimator. Since those predictions are out-of-sample, the scores represent valid test scores. The scorer should be a function that accepts an array of true values and an array of predictions: `score = f(y_true, y_pred)`.
- **raise_on_exception** (*bool (default = True)*) – whether to issue warnings on soft exceptions or raise error. Examples include lack of layers, bad inputs, and failed fit of an estimator in a layer. If set to `False`, warnings are issued instead but estimation continues unless exception is fatal. Note that this can result in unexpected behavior unless the exception is anticipated.
- **array_check** (*int (default = 2)*) – level of strictness in checking input arrays.
  - `array_check = 0` will not check `X` or `y`
  - `array_check = 1` will check `X` and `y` for inconsistencies and warn when format looks suspicious, but retain original format.
  - `array_check = 2` will impose Scikit-learn array checks, which converts `X` and `y` to numpy arrays and raises an error if conversion fails.
- **verbose** (*int or bool (default = False)*) – level of verbosity.
  - `verbose = 0` silent (same as `verbose = False`)
  - `verbose = 1` messages at start and finish (same as `verbose = True`)
  - `verbose = 2` messages for each layer

  If `verbose >= 50` prints to `sys.stdout`, else `sys.stderr`. For verbosity in the layers themselves, use `fit_params`.
- **n_jobs** (*int (default = -1)*) – number of CPU cores to use for fitting and prediction.
- **backend** (*str or object (default = 'threading')*) – backend infrastructure to use during call to `mlens.externals.joblib.Parallel`. See Joblib for further documentation. To set global backend, set `mlens.config.BACKEND`.

scores_

dict – if `scorer` was passed to instance, `scores_` contains dictionary with cross-validated scores assembled during `fit` call. The fold structure used for scoring is determined by `folds`.

Examples

Instantiate ensembles with no preprocessing: use list of estimators

```
>>> from mlens.ensemble import Subsemble
>>> from mlens.metrics.metrics import rmse
>>> from sklearn.datasets import load_boston
>>> from sklearn.linear_model import Lasso
>>> from sklearn.svm import SVR
>>>
>>> X, y = load_boston(True)
>>>
>>> ensemble = Subsemble()
>>> ensemble.add([SVR(), ('can name some or all est', Lasso())])
>>> ensemble.add(SVR(), meta=True)
>>>
>>> ensemble.fit(X, y)
>>> preds = ensemble.predict(X)
>>> rmse(y, preds)
9.2393246...
```

Instantiate ensembles with different preprocessing pipelines through dicts.

```
>>> from mlens.ensemble import Subsemble
>>> from mlens.metrics.metrics import rmse
>>> from sklearn.datasets import load_boston
>>> from sklearn. preprocessing import MinMaxScaler, StandardScaler
>>> from sklearn.linear_model import Lasso
>>> from sklearn.svm import SVR
>>>
>>> X, y = load_boston(True)
>>>
>>> preprocessing_cases = {'mm': [MinMaxScaler()],
...                        'sc': [StandardScaler()]}
>>>
>>> estimators_per_case = {'mm': [SVR()],
...                        'sc': [('can name some or all ests', Lasso())]}
>>>
>>> ensemble = Subsemble()
>>> ensemble.add(estimators_per_case, preprocessing_cases).add_meta(SVR())
>>>
>>> ensemble.fit(X, y)
>>> preds = ensemble.predict(X)
>>> rmse(y, preds)
9.0115741...
```

add(*estimators*, *preprocessing=None*, *meta=False*, *partitions=None*, *partition_estimator=None*, *folds=None*, *proba=False*, *propagate_features=None*, ***kwargs*)
Add layer to ensemble.

Parameters

- **preprocessing** (*dict of lists or list, optional (default = None)*) – preprocessing pipelines for given layer. If the same preprocessing applies to all estimators, `preprocessing` should be a list of transformer instances. The list can contain the instances directly, named tuples of transformers, or a combination of both.

```
option_1 = [transformer_1, transformer_2]
option_2 = [("trans-1", transformer_1),
```

```
            ("trans-2", transformer_2)]
option_3 = [transformer_1, ("trans-2", transformer_2)]
```

If different preprocessing pipelines are desired, a dictionary that maps preprocessing pipelines must be passed. The names of the preprocessing dictionary must correspond to the names of the estimator dictionary.

```
preprocessing_cases = {"case-1": [trans_1, trans_2],
                       "case-2": [alt_trans_1, alt_trans_2]}

estimators = {"case-1": [est_a, est_b],
              "case-2": [est_c, est_d]}
```

The lists for each dictionary entry can be any of `option_1`, `option_2` and `option_3`.

- **estimators** (*dict of lists or list or instance*) – estimators constituting the layer. If preprocessing is none and the layer is meant to be the meta estimator, it is permissible to pass a single instantiated estimator. If `preprocessing` is `None` or `list`, `estimators` should be a `list`. The list can either contain estimator instances, named tuples of estimator instances, or a combination of both.

```
option_1 = [estimator_1, estimator_2]
option_2 = [("est-1", estimator_1), ("est-2", estimator_2)]
option_3 = [estimator_1, ("est-2", estimator_2)]
```

If different preprocessing pipelines are desired, a dictionary that maps estimators to preprocessing pipelines must be passed. The names of the estimator dictionary must correspond to the names of the estimator dictionary.

```
preprocessing_cases = {"case-1": [trans_1, trans_2],
                       "case-2": [alt_trans_1, alt_trans_2]}

estimators = {"case-1": [est_a, est_b],
              "case-2": [est_c, est_d]}
```

The lists for each dictionary entry can be any of `option_1`, `option_2` and `option_3`.

- **meta** (*bool*) – indicator if the layer added is the final meta estimator. This will prevent folded or blended fits of the estimators and only fit them once on the full input data.
- **partitions** (*int, optional*) – number of partitions to split data into. Increasing partitions increases the number of estimators in the layer by a factor equal to the number of estimators. Specifying this parameter overrides the ensemble-wide parameter.
- **partition_estimator** (*instance, optional*) – To use a supervised or unsupervised estimator to learn partitions, pass an instantiated estimator as `partition_estimator`. The estimator must accept a `fit` call for fitting the training data, and a `predict` call that *assigns cluster partitions labels*. For instance, clustering estimator or classifiers (where class predictions will be used for partitioning). The number of partitions by the estimator must correspond to the layer's `partitions` argument. Passing an estimator here supersedes any other estimator previously passed.
- **folds** (*int, optional*) – Use if a different number of folds is desired than what the ensemble was instantiated with.
- **proba** (*bool (default = False)*) – whether to call `predict_proba` on base learners.

- **propagate_features** (*list, optional*) – List of column indexes to propagate from the input of the layer to the output of the layer. Propagated features are concatenated and stored in the leftmost columns of the output matrix. The `propagate_features` list should define a slice of the numpy array containing the input data, e.g. `[0, 1]` to propagate the first two columns of the input matrix to the output matrix.
- ****kwargs** (*optional*) – optional keyword arguments to instantiate ensemble with. In particular, keywords for clustered subsemble learning
  - **fit_estimator** *(Bool, default = True)* - whether to call `fit` on the partition estimator.
  - **attr** *(str, default = 'predict')* - the method attribute to call for generating partition ids for the input data.
  - **partition_on** *(str, default = 'X')* - the input data for the `attr` method. One of `'X'`, `'y'` or `'both'`.

Returns **self** – ensemble instance with layer instantiated.

Return type instance

add_meta(*estimator*, ***kwargs*)

Add meta estimator.

Parameters

- **estimator** (*instance*) – estimator instance.
- ****kwargs** (*optional*) – optional keyword arguments.

mlens.ensemble.super_learner module

ML-ENSEMBLE

author Sebastian Flennerhag

copyright 2017

licence MIT

Super Learner class. Fully integrable with Scikit-learn.

class mlens.ensemble.super_learner.**SuperLearner**(*folds=2*, *shuffle=False*, *random_state=None*, *scorer=None*, *raise_on_exception=True*, *array_check=2*, *verbose=False*, *n_jobs=-1*, *backend=None*, *layers=None*)

Bases: *mlens.ensemble.base.BaseEnsemble*

Super Learner class.

The Super Learner (also known as the Stacking Ensemble)is an supervised ensemble algorithm that uses K-fold estimation to map a training set (X, y) into a prediction set (Z, y), where the predictions in Z are constructed using K-Fold splits of X to ensure Z reflects test errors, and that applies a user-specified meta learner to predict y from Z. The algorithm in sudo code follows:

1. Specify a library L of base learners
2. Fit all base learners on X and store the fitted estimators.
3. Split X into K folds, fit every learner in L on the training set and predict test set. Repeat until all folds have been predicted.
4. Construct a matrix Z by stacking the predictions per fold.

5. Fit the meta learner on Z and store the learner

The ensemble can be used for prediction by mapping a new test set T into a prediction set Z' using the learners fitted in (2), and then mapping Z' to y' using the fitted meta learner from (5).

The Super Learner does asymptotically as well as (up to a constant) an Oracle selector. For the theory behind the Super Learner, see[1] and[2] as well as references therein.

Stacking K-fold predictions to cover an entire training set is a time consuming method and can be prohibitively costly for large datasets. With large data, other ensembles that fits an ensemble on subsets can achieve similar performance at a fraction of the training time. However, when data is noisy or of high variance, the *SuperLearner* ensure all information is used during fitting.

References

Notes

This implementation uses the agnostic meta learner approach, where the user supplies the meta learner to be used. For the original Super Learner algorithm (i.e. learn the best linear combination of the base learners), the user can specify a linear regression as the meta learner.

See also:

`BlendEnsemble`, `Subsemble`

Parameters

- **folds** (*int (default = 2)*) – number of folds to use during fitting. Note: this parameter can be specified on a layer-specific basis in the *add* method.
- **shuffle** (*bool (default = True)*) – whether to shuffle data before generating folds.
- **random_state** (*int (default = None)*) – random seed if shuffling inputs.
- **scorer** (*object (default = None)*) – scoring function. If a function is provided, base estimators will be scored on the training set assembled for fitting the meta estimator. Since those predictions are out-of-sample, the scores represent valid test scores. The scorer should be a function that accepts an array of true values and an array of predictions: `score = f(y_true, y_pred)`.
- **raise_on_exception** (*bool (default = True)*) – whether to issue warnings on soft exceptions or raise error. Examples include lack of layers, bad inputs, and failed fit of an estimator in a layer. If set to `False`, warnings are issued instead but estimation continues unless exception is fatal. Note that this can result in unexpected behavior unless the exception is anticipated.
- **array_check** (*int (default = 2)*) – level of strictness in checking input arrays.
  - `array_check = 0` will not check `X` or `y`
  - `array_check = 1` will check `X` and `y` for inconsistencies and warn when format looks suspicious, but retain original format.
  - `array_check = 2` will impose Scikit-learn array checks, which converts `X` and `y` to numpy arrays and raises an error if conversion fails.

[1] van der Laan, Mark J.; Polley, Eric C.; and Hubbard, Alan E., "Super Learner" (July 2007). U.C. Berkeley Division of Biostatistics Working Paper Series. Working Paper 222. http://biostats.bepress.com/ucbbiostat/paper222

[2] Polley, Eric C. and van der Laan, Mark J., "Super Learner In Prediction" (May 2010). U.C. Berkeley Division of Biostatistics Working Paper Series. Working Paper 266. http://biostats.bepress.com/ucbbiostat/paper266

- **verbose** (*int or bool (default = False)*) – level of verbosity.
  - `verbose = 0` silent (same as `verbose = False`)
  - `verbose = 1` messages at start and finish (same as `verbose = True`)
  - `verbose = 2` messages for each layer

  If `verbose >= 50` prints to `sys.stdout`, else `sys.stderr`. For verbosity in the layers themselves, use `fit_params`.
- **n_jobs** (*int (default = -1)*) – number of CPU cores to use for fitting and prediction.
- **backend** (*str or object (default = 'threading')*) – backend infrastructure to use during call to `mlens.externals.joblib.Parallel`. See Joblib for further documentation. To set global backend, set `mlens.config.BACKEND`.

scores_

dict – if `scorer` was passed to instance, `scores_` contains dictionary with cross-validated scores assembled during `fit` call. The fold structure used for scoring is determined by `folds`.

Examples

Instantiate ensembles with no preprocessing: use list of estimators

```
>>> from mlens.ensemble import SuperLearner
>>> from mlens.metrics.metrics import rmse
>>> from sklearn.datasets import load_boston
>>> from sklearn.linear_model import Lasso
>>> from sklearn.svm import SVR
>>>
>>> X, y = load_boston(True)
>>>
>>> ensemble = SuperLearner()
>>> ensemble.add([SVR(), ('can name some or all est', Lasso())])
>>> ensemble.add_meta(SVR())
>>>
>>> ensemble.fit(X, y)
>>> preds = ensemble.predict(X)
>>> rmse(y, preds)
6.955358...
```

Instantiate ensembles with different preprocessing pipelines through dicts.

```
>>> from mlens.ensemble import SuperLearner
>>> from mlens.metrics.metrics import rmse
>>> from sklearn.datasets import load_boston
>>> from sklearn. preprocessing import MinMaxScaler, StandardScaler
>>> from sklearn.linear_model import Lasso
>>> from sklearn.svm import SVR
>>>
>>> X, y = load_boston(True)
>>>
>>> preprocessing_cases = {'mm': [MinMaxScaler()],
...                        'sc': [StandardScaler()]}
>>>
>>> estimators_per_case = {'mm': [SVR()],
...                        'sc': [('can name some or all ests', Lasso())]}
```

```
>>>
>>> ensemble = SuperLearner()
>>> ensemble.add(estimators_per_case, preprocessing_cases).add(SVR(),
...                                                            meta=True)
>>>
>>> ensemble.fit(X, y)
>>> preds = ensemble.predict(X)
>>> rmse(y, preds)
7.841329...
```

add(*estimators*, *preprocessing=None*, *folds=None*, *proba=False*, *meta=False*, *propagate_features=None*, ***kwargs*)
Add layer to ensemble.

Parameters

- **estimators** (*dict of lists or list or instance*) – estimators constituting the layer. If preprocessing is none and the layer is meant to be the meta estimator, it is permissible to pass a single instantiated estimator. If `preprocessing` is `None` or `list`, `estimators` should be a `list`. The list can either contain estimator instances, named tuples of estimator instances, or a combination of both.

```
option_1 = [estimator_1, estimator_2]
option_2 = [("est-1", estimator_1), ("est-2", estimator_2)]
option_3 = [estimator_1, ("est-2", estimator_2)]
```

  If different preprocessing pipelines are desired, a dictionary that maps estimators to preprocessing pipelines must be passed. The names of the estimator dictionary must correspond to the names of the estimator dictionary.

```
preprocessing_cases = {"case-1": [trans_1, trans_2],
                       "case-2": [alt_trans_1, alt_trans_2]}

estimators = {"case-1": [est_a, est_b],
              "case-2": [est_c, est_d]}
```

  The lists for each dictionary entry can be any of `option_1`, `option_2` and `option_3`.

- **preprocessing** (*dict of lists or list, optional (default = None)*) – preprocessing pipelines for given layer. If the same preprocessing applies to all estimators, `preprocessing` should be a list of transformer instances. The list can contain the instances directly, named tuples of transformers, or a combination of both.

```
option_1 = [transformer_1, transformer_2]
option_2 = [("trans-1", transformer_1),
            ("trans-2", transformer_2)]
option_3 = [transformer_1, ("trans-2", transformer_2)]
```

  If different preprocessing pipelines are desired, a dictionary that maps preprocessing pipelines must be passed. The names of the preprocessing dictionary must correspond to the names of the estimator dictionary.

```
preprocessing_cases = {"case-1": [trans_1, trans_2],
                       "case-2": [alt_trans_1, alt_trans_2]}

estimators = {"case-1": [est_a, est_b],
              "case-2": [est_c, est_d]}
```

The lists for each dictionary entry can be any of `option_1`, `option_2` and `option_3`.

- **folds** (*int, optional*) – Use if a different number of folds is desired than what the ensemble was instantiated with.
- **proba** (*bool*) – whether layer should predict class probabilities. Note: setting `proba=True` will attempt to call an the estimators `predict_proba` method.
- **propagate_features** (*list, optional*) – List of column indexes to propagate from the input of the layer to the output of the layer. Propagated features are concatenated and stored in the leftmost columns of the output matrix. The `propagate_features` list should define a slice of the numpy array containing the input data, e.g. `[0, 1]` to propagate the first two columns of the input matrix to the output matrix.
- **meta** (*bool (default = False)*) – indicator if the layer added is the final meta estimator. This will prevent folded or blended fits of the estimators and only fit them once on the full input data.
- ****kwargs** (*optional*) – optional keyword arguments.

Returns **self** – ensemble instance with layer instantiated.

Return type instance

add_meta(*estimator*, ***kwargs*)
Meta Learner.

Meta learner to be used for final predictions.

Parameters

- **estimator** (*instance*) – estimator instance.
- ****kwargs** (*optional*) – optional keyword arguments.

Module contents

author Sebastian Flennerhag

copyright 2017

licence MIT

class mlens.ensemble.**SuperLearner**(*folds=2*, *shuffle=False*, *random_state=None*, *scorer=None*, *raise_on_exception=True*, *array_check=2*, *verbose=False*, *n_jobs=-1*, *backend=None*, *layers=None*)
Bases: *mlens.ensemble.base.BaseEnsemble*

Super Learner class.

The Super Learner (also known as the Stacking Ensemble)is an supervised ensemble algorithm that uses K-fold estimation to map a training set (X, y) into a prediction set (Z, y), where the predictions in Z are constructed using K-Fold splits of X to ensure Z reflects test errors, and that applies a user-specified meta learner to predict y from Z. The algorithm in sudo code follows:

1. Specify a library L of base learners
2. Fit all base learners on X and store the fitted estimators.
3. Split X into K folds, fit every learner in L on the training set and predict test set. Repeat until all folds have been predicted.
4. Construct a matrix Z by stacking the predictions per fold.

5. Fit the meta learner on Z and store the learner

The ensemble can be used for prediction by mapping a new test set T into a prediction set Z' using the learners fitted in (2), and then mapping Z' to y' using the fitted meta learner from (5).

The Super Learner does asymptotically as well as (up to a constant) an Oracle selector. For the theory behind the Super Learner, see[1] and[2] as well as references therein.

Stacking K-fold predictions to cover an entire training set is a time consuming method and can be prohibitively costly for large datasets. With large data, other ensembles that fits an ensemble on subsets can achieve similar performance at a fraction of the training time. However, when data is noisy or of high variance, the *SuperLearner* ensure all information is used during fitting.

References

Notes

This implementation uses the agnostic meta learner approach, where the user supplies the meta learner to be used. For the original Super Learner algorithm (i.e. learn the best linear combination of the base learners), the user can specify a linear regression as the meta learner.

See also:

BlendEnsemble, *Subsemble*

Parameters

- **folds** (*int (default = 2)*) – number of folds to use during fitting. Note: this parameter can be specified on a layer-specific basis in the *add* method.
- **shuffle** (*bool (default = True)*) – whether to shuffle data before generating folds.
- **random_state** (*int (default = None)*) – random seed if shuffling inputs.
- **scorer** (*object (default = None)*) – scoring function. If a function is provided, base estimators will be scored on the training set assembled for fitting the meta estimator. Since those predictions are out-of-sample, the scores represent valid test scores. The scorer should be a function that accepts an array of true values and an array of predictions: `score = f(y_true, y_pred)`.
- **raise_on_exception** (*bool (default = True)*) – whether to issue warnings on soft exceptions or raise error. Examples include lack of layers, bad inputs, and failed fit of an estimator in a layer. If set to `False`, warnings are issued instead but estimation continues unless exception is fatal. Note that this can result in unexpected behavior unless the exception is anticipated.
- **array_check** (*int (default = 2)*) – level of strictness in checking input arrays.
  - `array_check = 0` will not check `X` or `y`
  - `array_check = 1` will check `X` and `y` for inconsistencies and warn when format looks suspicious, but retain original format.
  - `array_check = 2` will impose Scikit-learn array checks, which converts `X` and `y` to numpy arrays and raises an error if conversion fails.

[1] van der Laan, Mark J.; Polley, Eric C.; and Hubbard, Alan E., "Super Learner" (July 2007). U.C. Berkeley Division of Biostatistics Working Paper Series. Working Paper 222. http://biostats.bepress.com/ucbbiostat/paper222

[2] Polley, Eric C. and van der Laan, Mark J., "Super Learner In Prediction" (May 2010). U.C. Berkeley Division of Biostatistics Working Paper Series. Working Paper 266. http://biostats.bepress.com/ucbbiostat/paper266

- **verbose** (*int or bool (default = False)*) – level of verbosity.
  - `verbose = 0` silent (same as `verbose = False`)
  - `verbose = 1` messages at start and finish (same as `verbose = True`)
  - `verbose = 2` messages for each layer

  If `verbose >= 50` prints to `sys.stdout`, else `sys.stderr`. For verbosity in the layers themselves, use `fit_params`.
- **n_jobs** (*int (default = -1)*) – number of CPU cores to use for fitting and prediction.
- **backend** (*str or object (default = 'threading')*) – backend infrastructure to use during call to `mlens.externals.joblib.Parallel`. See Joblib for further documentation. To set global backend, set `mlens.config.BACKEND`.

scores_

dict – if `scorer` was passed to instance, `scores_` contains dictionary with cross-validated scores assembled during `fit` call. The fold structure used for scoring is determined by `folds`.

Examples

Instantiate ensembles with no preprocessing: use list of estimators

```
>>> from mlens.ensemble import SuperLearner
>>> from mlens.metrics.metrics import rmse
>>> from sklearn.datasets import load_boston
>>> from sklearn.linear_model import Lasso
>>> from sklearn.svm import SVR
>>>
>>> X, y = load_boston(True)
>>>
>>> ensemble = SuperLearner()
>>> ensemble.add([SVR(), ('can name some or all est', Lasso())])
>>> ensemble.add_meta(SVR())
>>>
>>> ensemble.fit(X, y)
>>> preds = ensemble.predict(X)
>>> rmse(y, preds)
6.955358...
```

Instantiate ensembles with different preprocessing pipelines through dicts.

```
>>> from mlens.ensemble import SuperLearner
>>> from mlens.metrics.metrics import rmse
>>> from sklearn.datasets import load_boston
>>> from sklearn. preprocessing import MinMaxScaler, StandardScaler
>>> from sklearn.linear_model import Lasso
>>> from sklearn.svm import SVR
>>>
>>> X, y = load_boston(True)
>>>
>>> preprocessing_cases = {'mm': [MinMaxScaler()],
...                        'sc': [StandardScaler()]}
>>>
>>> estimators_per_case = {'mm': [SVR()],
...                        'sc': [('can name some or all ests', Lasso())]}
```

```
>>>
>>> ensemble = SuperLearner()
>>> ensemble.add(estimators_per_case, preprocessing_cases).add(SVR(),
...                                                            meta=True)
>>>
>>> ensemble.fit(X, y)
>>> preds = ensemble.predict(X)
>>> rmse(y, preds)
7.841329...
```

add(*estimators*, *preprocessing=None*, *folds=None*, *proba=False*, *meta=False*, *propagate_features=None*, ***kwargs*)
Add layer to ensemble.

Parameters

- **estimators** (*dict of lists or list or instance*) – estimators constituting the layer. If preprocessing is none and the layer is meant to be the meta estimator, it is permissible to pass a single instantiated estimator. If `preprocessing` is `None` or `list`, `estimators` should be a `list`. The list can either contain estimator instances, named tuples of estimator instances, or a combination of both.

```
option_1 = [estimator_1, estimator_2]
option_2 = [("est-1", estimator_1), ("est-2", estimator_2)]
option_3 = [estimator_1, ("est-2", estimator_2)]
```

  If different preprocessing pipelines are desired, a dictionary that maps estimators to preprocessing pipelines must be passed. The names of the estimator dictionary must correspond to the names of the estimator dictionary.

```
preprocessing_cases = {"case-1": [trans_1, trans_2],
                       "case-2": [alt_trans_1, alt_trans_2]}

estimators = {"case-1": [est_a, est_b],
              "case-2": [est_c, est_d]}
```

  The lists for each dictionary entry can be any of `option_1`, `option_2` and `option_3`.

- **preprocessing** (*dict of lists or list, optional (default = None)*) – preprocessing pipelines for given layer. If the same preprocessing applies to all estimators, `preprocessing` should be a list of transformer instances. The list can contain the instances directly, named tuples of transformers, or a combination of both.

```
option_1 = [transformer_1, transformer_2]
option_2 = [("trans-1", transformer_1),
            ("trans-2", transformer_2)]
option_3 = [transformer_1, ("trans-2", transformer_2)]
```

  If different preprocessing pipelines are desired, a dictionary that maps preprocessing pipelines must be passed. The names of the preprocessing dictionary must correspond to the names of the estimator dictionary.

```
preprocessing_cases = {"case-1": [trans_1, trans_2],
                       "case-2": [alt_trans_1, alt_trans_2]}

estimators = {"case-1": [est_a, est_b],
              "case-2": [est_c, est_d]}
```

The lists for each dictionary entry can be any of `option_1`, `option_2` and `option_3`.

- **folds** (*int, optional*) – Use if a different number of folds is desired than what the ensemble was instantiated with.
- **proba** (*bool*) – whether layer should predict class probabilities. Note: setting `proba=True` will attempt to call an the estimators `predict_proba` method.
- **propagate_features** (*list, optional*) – List of column indexes to propagate from the input of the layer to the output of the layer. Propagated features are concatenated and stored in the leftmost columns of the output matrix. The `propagate_features` list should define a slice of the numpy array containing the input data, e.g. `[0, 1]` to propagate the first two columns of the input matrix to the output matrix.
- **meta** (*bool (default = False)*) – indicator if the layer added is the final meta estimator. This will prevent folded or blended fits of the estimators and only fit them once on the full input data.
- ****kwargs** (*optional*) – optional keyword arguments.

Returns **self** – ensemble instance with layer instantiated.

Return type instance

add_meta(*estimator*, ***kwargs*)

Meta Learner.

Meta learner to be used for final predictions.

Parameters

- **estimator** (*instance*) – estimator instance.
- ****kwargs** (*optional*) – optional keyword arguments.

class mlens.ensemble.**BlendEnsemble**(*test_size=0.5*, *shuffle=False*, *random_state=None*, *scorer=None*, *raise_on_exception=True*, *array_check=2*, *verbose=False*, *n_jobs=-1*, *backend=None*, *layers=None*)

Bases: *mlens.ensemble.base.BaseEnsemble*

Blend Ensemble class.

The Blend Ensemble is a supervised ensemble closely related to the *SuperLearner*. It differs in that to estimate the prediction matrix Z used by the meta learner, it uses a subset of the data to predict its complement, and the meta learner is fitted on those predictions.

By only fitting every base learner once on a subset of the full training data, *BlendEnsemble* is a fast ensemble that can handle very large datasets simply by only using portion of it at each stage. The cost of this approach is that information is thrown out at each stage, as one layer will not see the training data used by the previous layer.

With large data that can be expected to satisfy an i.i.d. assumption, the *BlendEnsemble* can achieve similar performance to more sophisticated ensembles at a fraction of the training time. However, with data data is not uniformly distributed or exhibits high variance the *BlendEnsemble* can be a poor choice as information is lost at each stage of fitting.

See also:

SuperLearner, *Subsemble*

Parameters

- **test_size** (*int, float (default = 0.5)*) – the size of the test set for each layer. This parameter can be overridden in the `add` method if different test sizes is desired

for each layer. If a `float` is specified, it is presumed to be the fraction of the available data to be used for training, and so `0. < test_size < 1.`.

- **shuffle** (*bool (default = True)*) – whether to shuffle data before selecting training data.
- **random_state** (*int (default = None)*) – random seed if shuffling inputs.
- **scorer** (*object (default = None)*) – scoring function. If a function is provided, base estimators will be scored on the prediction made. The scorer should be a function that accepts an array of true values and an array of predictions: `score = f(y_true, y_pred)`.
- **raise_on_exception** (*bool (default = True)*) – whether to issue warnings on soft exceptions or raise error. Examples include lack of layers, bad inputs, and failed fit of an estimator in a layer. If set to `False`, warnings are issued instead but estimation continues unless exception is fatal. Note that this can result in unexpected behavior unless the exception is anticipated.
- **array_check** (*int (default = 2)*) – level of strictness in checking input arrays.
  - `array_check = 0` will not check `X` or `y`
  - `array_check = 1` will check `X` and `y` for inconsistencies and warn when format looks suspicious, but retain original format.
  - `array_check = 2` will impose Scikit-learn array checks, which converts `X` and `y` to numpy arrays and raises an error if conversion fails.
- **verbose** (*int or bool (default = False)*) – level of verbosity.
  - `verbose = 0` silent (same as `verbose = False`)
  - `verbose = 1` messages at start and finish (same as `verbose = True`)
  - `verbose = 2` messages for each layer

  If `verbose >= 50` prints to `sys.stdout`, else `sys.stderr`. For verbosity in the layers themselves, use `fit_params`.
- **n_jobs** (*int (default = -1)*) – number of CPU cores to use for fitting and prediction.
- **backend** (*str or object (default = 'threading')*) – backend infrastructure to use during call to `mlens.externals.joblib.Parallel`. See Joblib for further documentation. To set global backend, set `mlens.config.BACKEND`.

scores_

dict – if `scorer` was passed to instance, `scores_` contains dictionary with cross-validated scores assembled during `fit` call. The fold structure used for scoring is determined by `folds`.

Examples

Instantiate ensembles with no preprocessing: use list of estimators

```
>>> from mlens.ensemble import BlendEnsemble
>>> from mlens.metrics.metrics import rmse
>>> from sklearn.datasets import load_boston
>>> from sklearn.linear_model import Lasso
>>> from sklearn.svm import SVR
>>>
```

```
>>> X, y = load_boston(True)
>>>
>>> ensemble = BlendEnsemble()
>>> ensemble.add([SVR(), ('can name some or all est', Lasso())])
>>> ensemble.add_meta(SVR())
>>>
>>> ensemble.fit(X, y)
>>> preds = ensemble.predict(X)
>>> rmse(y, preds)
7.656098...
```

Instantiate ensembles with different preprocessing pipelines through dicts.

```
>>> from mlens.ensemble import BlendEnsemble
>>> from mlens.metrics.metrics import rmse
>>> from sklearn.datasets import load_boston
>>> from sklearn. preprocessing import MinMaxScaler, StandardScaler
>>> from sklearn.linear_model import Lasso
>>> from sklearn.svm import SVR
>>>
>>> X, y = load_boston(True)
>>>
>>> preprocessing_cases = {'mm': [MinMaxScaler()],
...                        'sc': [StandardScaler()]}
>>>
>>> estimators_per_case = {'mm': [SVR()],
...                        'sc': [('can name some or all ests', Lasso())]}
>>>
>>> ensemble = BlendEnsemble()
>>> ensemble.add(estimators_per_case, preprocessing_cases).add(SVR(),
...                                                            meta=True)
>>>
>>> ensemble.fit(X, y)
>>> preds = ensemble.predict(X)
>>> rmse(y, preds)
7.9814242...
```

add(*estimators*, *preprocessing=None*, *test_size=None*, *proba=False*, *meta=False*, *propagate_features=None*, ***kwargs*)
Add layer to ensemble.

Parameters

- **preprocessing** (*dict of lists or list, optional (default = None)*) – preprocessing pipelines for given layer. If the same preprocessing applies to all estimators, `preprocessing` should be a list of transformer instances. The list can contain the instances directly, named tuples of transformers, or a combination of both.

```
option_1 = [transformer_1, transformer_2]
option_2 = [("trans-1", transformer_1),
            ("trans-2", transformer_2)]
option_3 = [transformer_1, ("trans-2", transformer_2)]
```

If different preprocessing pipelines are desired, a dictionary that maps preprocessing pipelines must be passed. The names of the preprocessing dictionary must correspond to the names of the estimator dictionary.

```
preprocessing_cases = {"case-1": [trans_1, trans_2],
                       "case-2": [alt_trans_1, alt_trans_2]}

estimators = {"case-1": [est_a, est_b],
              "case-2": [est_c, est_d]}
```

The lists for each dictionary entry can be any of `option_1`, `option_2` and `option_3`.

- **estimators** (*dict of lists or list or instance*) – estimators constituting the layer. If preprocessing is none and the layer is meant to be the meta estimator, it is permissible to pass a single instantiated estimator. If `preprocessing` is `None` or `list`, `estimators` should be a `list`. The list can either contain estimator instances, named tuples of estimator instances, or a combination of both.

```
option_1 = [estimator_1, estimator_2]
option_2 = [("est-1", estimator_1), ("est-2", estimator_2)]
option_3 = [estimator_1, ("est-2", estimator_2)]
```

If different preprocessing pipelines are desired, a dictionary that maps estimators to preprocessing pipelines must be passed. The names of the estimator dictionary must correspond to the names of the estimator dictionary.

```
preprocessing_cases = {"case-1": [trans_1, trans_2],
                       "case-2": [alt_trans_1, alt_trans_2]}

estimators = {"case-1": [est_a, est_b],
              "case-2": [est_c, est_d]}
```

The lists for each dictionary entry can be any of `option_1`, `option_2` and `option_3`.

- **test_size** (*int or float, optional*) – Use if a different test set size is desired for layer than what the ensemble was instantiated with.
- **proba** (*bool (default = False)*) – Whether to call `predict_proba` on base learners.
- **propagate_features** (*list, optional*) – List of column indexes to propagate from the input of the layer to the output of the layer. Propagated features are concatenated and stored in the leftmost columns of the output matrix. The `propagate_features` list should define a slice of the numpy array containing the input data, e.g. `[0, 1]` to propagate the first two columns of the input matrix to the output matrix.
- **meta** (*bool (default = False)*) – Whether the layer should be treated as the final meta estimator.
- ****kwargs** (*optional*) – optional keyword arguments to instantiate layer with.

Returns **self** – ensemble instance with layer instantiated.

Return type instance

add_meta(*estimator*, ***kwargs*)

Meta Learner.

Compatibility method for adding a meta learner to be used for final predictions.

Parameters

- **estimator** (*instance*) – estimator instance.
- ****kwargs** (*optional*) – optional keyword arguments.

class mlens.ensemble.**Subsemble**(*partitions=2*, *partition_estimator=None*, *folds=2*, *shuffle=False*, *random_state=None*, *scorer=None*, *raise_on_exception=True*, *array_check=2*, *verbose=False*, *n_jobs=-1*, *backend=None*, *layers=None*)

Bases: *mlens.ensemble.base.BaseEnsemble*

Subsemble class.

Subsemble is a supervised ensemble algorithm that uses subsets of the full data to fit a layer, and within each subset K-fold estimation to map a training set (X, y) into a prediction set (Z, y), where Z is a matrix of prediction from each estimator on each subset (thus of shape `[n_samples, (n_partitions * n_estimators)]`). Z is constructed using K-Fold splits of each partition of *X* to ensure Z reflects test errors within each partition. A final user-specified meta learner is fitted to the final ensemble layer's prediction, to learn the best combination of subset-specific estimator predictions. By passing a `partition_estimator`, the partitions can be learnt. The algorithm in sudo code :

1. For each layer in the ensemble, do:
   (a) Specify a library of L base learners
   (b) Specify a partition strategy and partition X into J subsets.
   (c) For each partition do:
      i. Fit all base learners and store them
      ii. Create K folds
      iii. For each fold, do:
         A. Fit all base learners on the training folds
         B. Collect *all* test folds, across partitions, and predict.
   (d) Assemble a cross-validated prediction matrix $Z \in \mathbb{R}^{(n \times (L \times J))}$ by stacking predictions made in the cross-validation step.
2. Fit the meta learner on Z and store the learner.

The ensemble can be used for prediction by mapping a new test set T into a prediction set Z' using the learners fitted in (1.3.1), and then using Z' to generate final predictions through the fitted meta learner from (2).

The Subsemble does asymptotically as well as (up to a constant) the Oracle selector. For the theory behind the Subsemble, see[3] and references therein.

By partitioning the data into subset and fitting on those, a Subsemble can reduce training time considerably if estimators does not scale linearly. Moreover, Subsemble allows estimators to learn different patterns from each subset, and so can improve the overall performance by achieving a tighter fit on each subset. Since all observations in the training set are predicted, no information is lost between layers.

This implementation allows very general partition estimators. The user must ensure that the partition estimator behaves as desired. To alter the expected behavior, see the `kwd` parameter under the `add` method and the *mlens.base.ClusteredSubsetIndex*. Also see the advanced tutorials for example use cases.

References

See also:

BlendEnsemble, *SuperLearner*

[3] Sapp, S., van der Laan, M. J., & Canny, J. (2014). Subsemble: an ensemble method for combining subset-specific algorithm fits. Journal of Applied Statistics, 41(6), 1247-1259. http://doi.org/10.1080/02664763.2013.864263

Parameters

- **partitions** (*int (default = 2)*) – number of partitions to split data into. For each layer, increasing partitions increases the number of estimators in the ensemble by a factor equal to the number of estimators. Note: this parameter can be specified on a layer-specific basis in the *add* method.
- **partition_estimator** (*instance, optional*) – To use a supervised or unsupervised estimator to learn partitions, pass an instantiated estimator as `partition_estimator`. The estimator must accept a `fit` call for fitting the training data, and a `predict` call that *assigns cluster partitions labels*. For instance, clustering estimator or classifiers (where their class predictions will be used for partitioning). The number of partitions by the estimator must correspond to the `partitions` argument. Specific estimators can be added to each layer by passing the estimator during the call to the ensemble's `add` method.
- **folds** (*int (default = 2)*) – number of folds to use during fitting. Note: this parameter can be specified on a layer-specific basis in the *add* method.
- **shuffle** (*bool (default = True)*) – whether to shuffle data before generating folds.
- **random_state** (*int (default = None)*) – random seed if shuffling inputs.
- **scorer** (*object (default = None)*) – scoring function. If a function is provided, base estimators will be scored on the training set assembled for fitting the meta estimator. Since those predictions are out-of-sample, the scores represent valid test scores. The scorer should be a function that accepts an array of true values and an array of predictions: `score = f(y_true, y_pred)`.
- **raise_on_exception** (*bool (default = True)*) – whether to issue warnings on soft exceptions or raise error. Examples include lack of layers, bad inputs, and failed fit of an estimator in a layer. If set to `False`, warnings are issued instead but estimation continues unless exception is fatal. Note that this can result in unexpected behavior unless the exception is anticipated.
- **array_check** (*int (default = 2)*) – level of strictness in checking input arrays.
  - `array_check = 0` will not check `X` or `y`
  - `array_check = 1` will check `X` and `y` for inconsistencies and warn when format looks suspicious, but retain original format.
  - `array_check = 2` will impose Scikit-learn array checks, which converts `X` and `y` to numpy arrays and raises an error if conversion fails.
- **verbose** (*int or bool (default = False)*) – level of verbosity.
  - `verbose = 0` silent (same as `verbose = False`)
  - `verbose = 1` messages at start and finish (same as `verbose = True`)
  - `verbose = 2` messages for each layer

  If `verbose >= 50` prints to `sys.stdout`, else `sys.stderr`. For verbosity in the layers themselves, use `fit_params`.
- **n_jobs** (*int (default = -1)*) – number of CPU cores to use for fitting and prediction.
- **backend** (*str or object (default = 'threading')*) – backend infrastructure to use during call to `mlens.externals.joblib.Parallel`. See Joblib for further documentation. To set global backend, set `mlens.config.BACKEND`.

scores_
dict – if `scorer` was passed to instance, `scores_` contains dictionary with cross-validated scores assembled during `fit` call. The fold structure used for scoring is determined by `folds`.

Examples

Instantiate ensembles with no preprocessing: use list of estimators

```
>>> from mlens.ensemble import Subsemble
>>> from mlens.metrics.metrics import rmse
>>> from sklearn.datasets import load_boston
>>> from sklearn.linear_model import Lasso
>>> from sklearn.svm import SVR
>>>
>>> X, y = load_boston(True)
>>>
>>> ensemble = Subsemble()
>>> ensemble.add([SVR(), ('can name some or all est', Lasso())])
>>> ensemble.add(SVR(), meta=True)
>>>
>>> ensemble.fit(X, y)
>>> preds = ensemble.predict(X)
>>> rmse(y, preds)
9.2393246...
```

Instantiate ensembles with different preprocessing pipelines through dicts.

```
>>> from mlens.ensemble import Subsemble
>>> from mlens.metrics.metrics import rmse
>>> from sklearn.datasets import load_boston
>>> from sklearn. preprocessing import MinMaxScaler, StandardScaler
>>> from sklearn.linear_model import Lasso
>>> from sklearn.svm import SVR
>>>
>>> X, y = load_boston(True)
>>>
>>> preprocessing_cases = {'mm': [MinMaxScaler()],
...                        'sc': [StandardScaler()]}
>>>
>>> estimators_per_case = {'mm': [SVR()],
...                        'sc': [('can name some or all ests', Lasso())]}
>>>
>>> ensemble = Subsemble()
>>> ensemble.add(estimators_per_case, preprocessing_cases).add_meta(SVR())
>>>
>>> ensemble.fit(X, y)
>>> preds = ensemble.predict(X)
>>> rmse(y, preds)
9.0115741...
```

add(*estimators*, *preprocessing=None*, *meta=False*, *partitions=None*, *partition_estimator=None*, *folds=None*, *proba=False*, *propagate_features=None*, ***kwargs*)
Add layer to ensemble.

Parameters

- **preprocessing** (*dict of lists or list, optional (default = None)*) – preprocessing pipelines for given layer. If the same preprocessing applies to

all estimators, `preprocessing` should be a list of transformer instances. The list can contain the instances directly, named tuples of transformers, or a combination of both.

```
option_1 = [transformer_1, transformer_2]
option_2 = [("trans-1", transformer_1),
            ("trans-2", transformer_2)]
option_3 = [transformer_1, ("trans-2", transformer_2)]
```

If different preprocessing pipelines are desired, a dictionary that maps preprocessing pipelines must be passed. The names of the preprocessing dictionary must correspond to the names of the estimator dictionary.

```
preprocessing_cases = {"case-1": [trans_1, trans_2],
                       "case-2": [alt_trans_1, alt_trans_2]}

estimators = {"case-1": [est_a, est_b],
              "case-2": [est_c, est_d]}
```

The lists for each dictionary entry can be any of `option_1`, `option_2` and `option_3`.

- **estimators** (*dict of lists or list or instance*) – estimators constituting the layer. If preprocessing is none and the layer is meant to be the meta estimator, it is permissible to pass a single instantiated estimator. If `preprocessing` is `None` or `list`, `estimators` should be a `list`. The list can either contain estimator instances, named tuples of estimator instances, or a combination of both.

```
option_1 = [estimator_1, estimator_2]
option_2 = [("est-1", estimator_1), ("est-2", estimator_2)]
option_3 = [estimator_1, ("est-2", estimator_2)]
```

If different preprocessing pipelines are desired, a dictionary that maps estimators to preprocessing pipelines must be passed. The names of the estimator dictionary must correspond to the names of the estimator dictionary.

```
preprocessing_cases = {"case-1": [trans_1, trans_2],
                       "case-2": [alt_trans_1, alt_trans_2]}

estimators = {"case-1": [est_a, est_b],
              "case-2": [est_c, est_d]}
```

The lists for each dictionary entry can be any of `option_1`, `option_2` and `option_3`.

- **meta** (*bool*) – indicator if the layer added is the final meta estimator. This will prevent folded or blended fits of the estimators and only fit them once on the full input data.
- **partitions** (*int, optional*) – number of partitions to split data into. Increasing partitions increases the number of estimators in the layer by a factor equal to the number of estimators. Specifying this parameter overrides the ensemble-wide parameter.
- **partition_estimator** (*instance, optional*) – To use a supervised or unsupervised estimator to learn partitions, pass an instantiated estimator as `partition_estimator`. The estimator must accept a `fit` call for fitting the training data, and a `predict` call that *assigns cluster partitions labels*. For instance, clustering estimator or classifiers (where class predictions will be used for partitioning). The number of partitions by the estimator must correspond to the layer's `partitions` argument. Passing an estimator here supersedes any other estimator previously passed.
- **folds** (*int, optional*) – Use if a different number of folds is desired than what the ensemble was instantiated with.

- **proba** (*bool (default = False)*) – whether to call `predict_proba` on base learners.
- **propagate_features** (*list, optional*) – List of column indexes to propagate from the input of the layer to the output of the layer. Propagated features are concatenated and stored in the leftmost columns of the output matrix. The `propagate_features` list should define a slice of the numpy array containing the input data, e.g. `[0, 1]` to propagate the first two columns of the input matrix to the output matrix.
- ****kwargs** (*optional*) – optional keyword arguments to instantiate ensemble with. In particular, keywords for clustered subsemble learning
  - **fit_estimator** *(Bool, default = True)* - whether to call `fit` on the partition estimator.
  - **attr** *(str, default = 'predict')* - the method attribute to call for generating partition ids for the input data.
  - **partition_on** *(str, default = 'X')* - the input data for the `attr` method. One of `'X'`, `'y'` or `'both'`.

Returns **self** – ensemble instance with layer instantiated.

Return type instance

add_meta(*estimator*, ***kwargs*)

Add meta estimator.

Parameters

- **estimator** (*instance*) – estimator instance.
- ****kwargs** (*optional*) – optional keyword arguments.

class mlens.ensemble.**SequentialEnsemble**(*shuffle=False*, *random_state=None*, *scorer=None*, *raise_on_exception=True*, *array_check=2*, *verbose=False*, *n_jobs=-1*, *backend=None*, *layers=None*)

Bases: *mlens.ensemble.base.BaseEnsemble*

Sequential Ensemble class.

The Sequential Ensemble class allows users to build ensembles with different classes of layers. The type of layer and its parameters are specified when added to the ensemble. See respective ensemble class for details on parameters.

See also:

BlendEnsemble, *Subsemble*, *SuperLearner*

Parameters

- **shuffle** (*bool (default = True)*) – whether to shuffle data before generating folds.
- **random_state** (*int (default = None)*) – random seed if shuffling inputs.
- **scorer** (*object (default = None)*) – scoring function. If a function is provided, base estimators will be scored on the training set assembled for fitting the meta estimator. Since those predictions are out-of-sample, the scores represent valid test scores. The scorer should be a function that accepts an array of true values and an array of predictions: `score = f(y_true, y_pred)`.
- **raise_on_exception** (*bool (default = True)*) – whether to issue warnings on soft exceptions or raise error. Examples include lack of layers, bad inputs, and failed

fit of an estimator in a layer. If set to `False`, warnings are issued instead but estimation continues unless exception is fatal. Note that this can result in unexpected behavior unless the exception is anticipated.

- **array_check** (*int (default = 2)*) – level of strictness in checking input arrays.
  - `array_check = 0` will not check `X` or `y`
  - `array_check = 1` will check `X` and `y` for inconsistencies and warn when format looks suspicious, but retain original format.
  - `array_check = 2` will impose Scikit-learn array checks, which converts `X` and `y` to numpy arrays and raises an error if conversion fails.
- **verbose** (*int or bool (default = False)*) – level of verbosity.
  - `verbose = 0` silent (same as `verbose = False`)
  - `verbose = 1` messages at start and finish (same as `verbose = True`)
  - `verbose = 2` messages for each layer

  If `verbose >= 50` prints to `sys.stdout`, else `sys.stderr`. For verbosity in the layers themselves, use `fit_params`.
- **n_jobs** (*int (default = -1)*) – number of CPU cores to use for fitting and prediction.
- **backend** (*str or object (default = 'threading')*) – backend infrastructure to use during call to `mlens.externals.joblib.Parallel`. See Joblib for further documentation. To change global backend, set `mlens.config.BACKEND`

scores_

dict – if `scorer` was passed to instance, `scores_` contains dictionary with cross-validated scores assembled during `fit` call. The fold structure used for scoring is determined by `folds`.

Examples

```
>>> from mlens.ensemble import SequentialEnsemble
>>> from mlens.metrics.metrics import rmse
>>> from sklearn.datasets import load_boston
>>> from sklearn.linear_model import Lasso
>>> from sklearn.svm import SVR
>>>
>>> X, y = load_boston(True)
>>>
>>> ensemble = SequentialEnsemble()
>>>
>>> # Add a subsemble with 5 partitions as first layer
>>> ensemble.add('subset', [SVR(), Lasso()], n_partitions=10, n_splits=10)
>>>
>>> # Add a super learner as second layer
>>> ensemble.add('stack', [SVR(), Lasso()], n_splits=20)
>>>
>>> # Specify a meta estimator
>>> ensemble.add_meta(SVR())
>>>
>>> ensemble.fit(X, y)
>>> preds = ensemble.predict(X)
```

```
>>> rmse(y, preds)
6.5628...
```

add(*cls*, *estimators*, *preprocessing=None*, ***kwargs*)
Add layer to ensemble.

For full set of optional arguments, see the ensemble API for the specified type.

Parameters

- **cls** (*str*) – layer class. Accepted types are:
  - 'blend' : blend ensemble
  - 'subset' : subsemble
  - 'stack' : super learner
- **estimators** (*dict of lists or list or instance*) – estimators constituting the layer. If preprocessing is none and the layer is meant to be the meta estimator, it is permissible to pass a single instantiated estimator. If `preprocessing` is `None` or `list`, `estimators` should be a `list`. The list can either contain estimator instances, named tuples of estimator instances, or a combination of both.

```
option_1 = [estimator_1, estimator_2]
option_2 = [("est-1", estimator_1), ("est-2", estimator_2)]
option_3 = [estimator_1, ("est-2", estimator_2)]
```

If different preprocessing pipelines are desired, a dictionary that maps estimators to preprocessing pipelines must be passed. The names of the estimator dictionary must correspond to the names of the estimator dictionary.

```
preprocessing_cases = {"case-1": [trans_1, trans_2],
                       "case-2": [alt_trans_1, alt_trans_2]}

estimators = {"case-1": [est_a, est_b],
              "case-2": [est_c, est_d]}
```

The lists for each dictionary entry can be any of `option_1`, `option_2` and `option_3`.

- **preprocessing** (*dict of lists or list, optional (default = None)*) – preprocessing pipelines for given layer. If the same preprocessing applies to all estimators, `preprocessing` should be a list of transformer instances. The list can contain the instances directly, named tuples of transformers, or a combination of both.

```
option_1 = [transformer_1, transformer_2]
option_2 = [("trans-1", transformer_1),
            ("trans-2", transformer_2)]
option_3 = [transformer_1, ("trans-2", transformer_2)]
```

If different preprocessing pipelines are desired, a dictionary that maps preprocessing pipelines must be passed. The names of the preprocessing dictionary must correspond to the names of the estimator dictionary.

```
preprocessing_cases = {"case-1": [trans_1, trans_2],
                       "case-2": [alt_trans_1, alt_trans_2]}

estimators = {"case-1": [est_a, est_b],
              "case-2": [est_c, est_d]}
```

The lists for each dictionary entry can be any of `option_1`, `option_2` and `option_3`.

- ****kwargs** (*optional*) – optional keyword arguments to instantiate layer with. See respective ensemble for further details.

Returns **self** – ensemble instance with layer instantiated.

Return type instance

add_meta(*estimator*, ***kwargs*)
Meta Learner.

Meta learner to be used for final predictions.

Parameters

- **estimator** (*instance*) – estimator instance.
- ****kwargs** (*optional*) – optional keyword arguments.

mlens.metrics package

Submodules

mlens.metrics.metrics module

ML-ENSEMBLE

author Sebastian Flennerhag

copyright 2017

license MIT

Scoring functions.

mlens.metrics.metrics.**mape**(*y*, *p*)
Mean Average Percentage Error.

$$MAPE(\mathbf{y}, \mathbf{p}) = |S| \sum_{i \in S} |\frac{y_i - p_i}{y_i}|$$

Parameters

- **y** (*array-like of shape [n_samples,]*) – ground truth.
- **p** (*array-like of shape [n_samples,]*) – predicted labels.

Returns **z** – mean average percentage error.

Return type float

mlens.metrics.metrics.**rmse**(*y*, *p*)
Root Mean Square Error.

$$RMSE(\mathbf{y}, \mathbf{p}) = \sqrt{MSE(\mathbf{y}, \mathbf{p})},$$

with

$$MSE(\mathbf{y}, \mathbf{p}) = |S| \sum_{i \in S} (y_i - p_i)^2$$

Parameters

- **y** (*array-like of shape [n_samples,]*) – ground truth.
- **p** (*array-like of shape [n_samples,]*) – predicted labels.

Returns **z** – root mean squared error.

Return type float

mlens.metrics.metrics.**wape**(*y*, *p*)

Weighted Mean Average Percentage Error.

$$WAPE(\mathbf{y}, \mathbf{p}) = \frac{\sum_{i \in S} |y_i - p_i|}{\sum_{i \in S} |y_i|}$$

Parameters

- **y** (*array-like of shape [n_samples,]*) – ground truth.
- **p** (*array-like of shape [n_samples,]*) – predicted labels.

Returns **z** – weighted mean average percentage error.

Return type float

Module contents

ML-ENSEMBLE

author Sebastian Flennerhag

copyright 2017

license MIT

mlens.metrics.**rmse**(*y*, *p*)

Root Mean Square Error.

$$RMSE(\mathbf{y}, \mathbf{p}) = \sqrt{MSE(\mathbf{y}, \mathbf{p})},$$

with

$$MSE(\mathbf{y}, \mathbf{p}) = |S| \sum_{i \in S} (y_i - p_i)^2$$

Parameters

- **y** (*array-like of shape [n_samples,]*) – ground truth.
- **p** (*array-like of shape [n_samples,]*) – predicted labels.

Returns **z** – root mean squared error.

Return type float

mlens.metrics.**mape**(*y*, *p*)

Mean Average Percentage Error.

$$MAPE(\mathbf{y}, \mathbf{p}) = |S| \sum_{i \in S} |\frac{y_i - p_i}{y_i}|$$

Parameters

- **y** (*array-like of shape [n_samples,]*) – ground truth.

- **p** (*array-like of shape [n_samples,]*) – predicted labels.

Returns **z** – mean average percentage error.

Return type float

mlens.metrics.**wape**(*y*, *p*)

Weighted Mean Average Percentage Error.

$$WAPE(\mathbf{y}, \mathbf{p}) = \frac{\sum_{i \in S} |y_i - p_i|}{\sum_{i \in S} |y_i|}$$

Parameters

- **y** (*array-like of shape [n_samples,]*) – ground truth.
- **p** (*array-like of shape [n_samples,]*) – predicted labels.

Returns **z** – weighted mean average percentage error.

Return type float

mlens.metrics.**make_scorer**(*score_func*, *greater_is_better=True*, *needs_proba=False*, *needs_threshold=False*, ***kwargs*)

Make a scorer from a performance metric or loss function.

This factory function wraps scoring functions for use in GridSearchCV and cross_val_score. It takes a score function, such as `accuracy_score`, `mean_squared_error`, `adjusted_rand_index` or `average_precision` and returns a callable that scores an estimator's output.

Read more in the User Guide.

Parameters

- **score_func** (*callable,*) – Score function (or loss function) with signature `score_func(y, y_pred, **kwargs)`.
- **greater_is_better** (*boolean, default=True*) – Whether score_func is a score function (default), meaning high is good, or a loss function, meaning low is good. In the latter case, the scorer object will sign-flip the outcome of the score_func.
- **needs_proba** (*boolean, default=False*) – Whether score_func requires predict_proba to get probability estimates out of a classifier.
- **needs_threshold** (*boolean, default=False*) – Whether score_func takes a continuous decision certainty. This only works for binary classification using estimators that have either a decision_function or predict_proba method.

  For example `average_precision` or the area under the roc curve can not be computed using discrete predictions alone.
- ****kwargs** (*additional arguments*) – Additional parameters to be passed to score_func.

Returns **scorer** – Callable object that returns a scalar score; greater is better.

Return type callable

Examples

```
>>> from sklearn.metrics import fbeta_score, make_scorer
>>> ftwo_scorer = make_scorer(fbeta_score, beta=2)
>>> ftwo_scorer
make_scorer(fbeta_score, beta=2)
>>> from sklearn.model_selection import GridSearchCV
>>> from sklearn.svm import LinearSVC
>>> grid = GridSearchCV(LinearSVC(), param_grid={'C': [1, 10]},
...                     scoring=ftwo_scorer)
```

mlens.model_selection package

Submodules

mlens.model_selection.model_selection module

ML-ENSEMBLE

author Sebastian Flennerhag

copyright 2017

licence MIT

Class for parallel tuning a set of estimators that share a common preprocessing pipeline.

class mlens.model_selection.model_selection.**Evaluator**(*scorer*, *cv=2*, *shuffle=True*, *random_state=None*, *backend=None*, *error_score=None*, *metrics=None*, *array_check=2*, *n_jobs=-1*, *verbose=False*)

Bases: object

Model selection across several estimators and preprocessing pipelines.

The *Evaluator* allows users to evaluate several models in one call across a set preprocessing pipelines. The class is useful for comparing a set of estimators, especially when several preprocessing pipelines is to be evaluated. By pre-making all folds and iteratively fitting estimators with different parameter settings, array slicing and preprocessing is kept to a minimum. This can greatly reduced fit time compared to creating pipeline classes for each estimator and pipeline and fitting them one at a time in an Scikit-learn `sklearn.model_selection.GridSearch` class.

Preprocessing can be done before making any evaluation, and several evaluations can be made on the pre-made folds. Current implementation relies on a randomized grid search, so parameter grids must be specified as SciPy distributions (or a class that accepts a `rvs` method).

Parameters

- **scorer** (*function*) – a scoring function that follows the Scikit-learn API:

```
score = scorer(estimator, y_true, y_pred)
```

A user defines scoring function, `score = f(y_true, y_pred)` can be made into a scorer by calling on the ML-Ensemble implementation of Scikit-learn's `make_scorer`. NOTE: do **not** use Scikit-learn's `make_scorer` if the Evaluator is to be pickled.

```
from mlens.metrics import make_scorer
scorer = make_scorer(scoring_function, **kwargs)
```

- **error_score** (*int, optional*) – score to assign when fitting an estimator fails. If `None`, the evaluator will raise an error.
- **cv** (*int or obj (default = 2)*) – cross validation folds to use. Either pass a `KFold` class that obeys the Scikit-learn API.
- **metrics** (*list, optional*) – list of aggregation metrics to calculate on scores. Default is mean and standard deviation.
- **shuffle** (*bool (default = True)*) – whether to shuffle input data before creating cv folds.
- **random_state** (*int, optional*) – seed for creating folds (if shuffled) and parameter draws
- **array_check** (*int (default = 2)*) – level of strictness in checking input arrays.
  - `array_check = 0` will not check `X` or `y`
  - `array_check = 1` will check `X` and `y` for inconsistencies and warn when format looks suspicious, but retain original format.
  - `array_check = 2` will impose Scikit-learn array checks, which converts `X` and `y` to numpy arrays and raises an error if conversion fails.
- **n_jobs** (*int (default = -1)*) – number of CPU cores to use.
- **verbose** (*bool or int (default = False)*) – level of printed messages.

summary
dict – Summary output that shows data for best mean test scores, such as test and train scores, std, fit times, and params.

cv_results
dict – a nested `dict` of data from each fit. Includes mean and std of test and train scores and fit times, as well as param draw index and parameters.

evaluate(*X*, *y*, *estimators*, *param_dicts*, *n_iter=2*)
Evaluate set of estimators.

Function for evaluating a set of estimators using cross validation. Similar to a randomized grid search, but applies the grid search to all specified preprocessing pipelines.

Parameters

- **X** (*array-like, shape=[n_samples, n_features]*) – input data to preprocess and create folds from.
- **y** (*array-like, shape=[n_samples,]*) – training labels.
- **estimators** (*list or dict*) – set of estimators to use. If no preprocessing is desired or if only on preprocessing pipeline should apply to all, pass a list of estimators. The list can contain elements of named tuples (i.e. `('my_name', my_est)`).

  If different estimators should be mapped to preprocessing cases, a dictionary that maps estimators to each case should be passed: `{'case_a': list_of_est, ...}`.
- **param_dicts** (*dict*) – parameter distribution mapping for estimators. Current implementation only supports randomized grid search. Passed distribution object must have an `rvs` method. See `Scipy.stats` for details.

  There is quite some flexibility in specifying `param_dicts`. If there is no preprocessing, or if all estimators are fitted on all preprocessing cases, the `param_dict` should have keys matching the names of the estimators.

```
estimators = [('name', est), est]

param_dicts = {'name': {'param-1': some_distribution},
               'est': {'param-1': some_distribution}
              }
```

It is possible to specify different distributions for some or all preprocessing cases:

```
preprocessing = {'case-1': transformer_list,
                 'case-2': transformer_list}

estimators = [('name', est), est]

param_dicts = {'name':
                   {'param-1': some_distribution},
               ('case-1', 'est'):
                   {'param-1': some_distribution}
               ('case-2', 'est'):
                   {'param-1': some_distribution,
                    'param-2': some_distribution}
              }
```

If estimators are mapped on a per-preprocessing case basis as a dictionary, `param_dict` must have key entries of the form `(case_name, est_name)`.

- **n_iter** (*int*) – number of parameter draws to evaluate.

Returns **self** – class instance with stored estimator evaluation results.

Return type instance

fit (*X*, *y*, *estimators*, *param_dicts*, *n_iter=2*, *preprocessing=None*)
Fit the Evaluator to given data, estimators and preprocessing.

Utility function that calls `preprocess` and `evaluate`. The following is equivalent:

```
# Explicitly calling preprocess and evaluate
evaluator.preprocess(X, y, preprocessing)
evaluator.evaluate(X, y, estimators, param_dicts, n_iter)

# Calling fit
evaluator.fit(X, y, estimators, param_dicts, n_iter, preprocessing)
```

Parameters

- **X** (*array-like, shape=[n_samples, n_features]*) – input data to preprocess and create folds from.
- **y** (*array-like, shape=[n_samples,]*) – training labels.
- **estimators** (*list or dict*) – set of estimators to use. If no preprocessing is desired or if only on preprocessing pipeline should apply to all, pass a list of estimators. The list can contain elements of named tuples (i.e. `('my_name', my_est)`).

  If different estimators should be mapped to preprocessing cases, a dictionary that maps estimators to each case should be passed: `{'case_a': list_of_est, ...}`.
- **param_dicts** (*dict*) – parameter distribution mapping for estimators. Current implementation only supports randomized grid search. Passed distribution object must have an `rvs` method. See `Scipy.stats` for details.

There is quite some flexibility in specifying `param_dicts`. If there is no preprocessing, or if all estimators are fitted on all preprocessing cases, the `param_dict` should have keys matching the names of the estimators.

```
estimators = [('name', est), est]

param_dicts = {'name': {'param-1': some_distribution},
               'est': {'param-1': some_distribution}
              }
```

It is possible to specify different distributions for some or all preprocessing cases:

```
preprocessing = {'case-1': transformer_list,
                 'case-2': transformer_list}

estimators = [('name', est), est]

param_dicts = {'name':
                   {'param-1': some_distribution},
               ('case-1', 'est'):
                   {'param-1': some_distribution}
               ('case-2', 'est'):
                   {'param-1': some_distribution,
                    'param-2': some_distribution}
              }
```

If estimators are mapped on a per-preprocessing case basis as a dictionary, `param_dict` must have key entries of the form `(case_name, est_name)`.

- **n_iter** (*int*) – number of parameter draws to evaluate.
- **preprocessing** (*dict, optional*) – preprocessing cases to consider. Pass a dictionary mapping a case name to a preprocessing pipeline.

```
preprocessing = {'case_name': transformer_list,}
```

Returns **self** – class instance with stored estimator evaluation results.

Return type instance

initialize(*X*, *y*)
Set up `ParallelEvaluation` job manager.

preprocess(*X*, *y*, *preprocessing=None*)
Preprocess folds.

Method for preprocessing data separately from the evaluation method. Helpful if preprocessing is costly relative to estimator fitting and several `evaluate` calls might be desired.

Parameters

- **X** (*array-like, shape=[n_samples, n_features]*) – input data to preprocess and create folds from.
- **y** (*array-like, shape=[n_samples,]*) – training labels.
- **preprocessing** (*list or dict, optional*) – preprocessing cases to consider. Pass a dictionary mapping a case name to a preprocessing pipeline.

```
preprocessing = {'case_name': transformer_list,}
```

Returns **self** – class instance with stored estimator evaluation results.

Return type instance

terminate()
Terminate evaluation job.

Module contents

author *Sebastian Flennerhag*

copyright *2017*

licence *MIT*

class mlens.model_selection.**Evaluator**(*scorer*, *cv=2*, *shuffle=True*, *random_state=None*, *backend=None*, *error_score=None*, *metrics=None*, *array_check=2*, *n_jobs=-1*, *verbose=False*)

Bases: object

Model selection across several estimators and preprocessing pipelines.

The *Evaluator* allows users to evaluate several models in one call across a set preprocessing pipelines. The class is useful for comparing a set of estimators, especially when several preprocessing pipelines is to be evaluated. By pre-making all folds and iteratively fitting estimators with different parameter settings, array slicing and preprocessing is kept to a minimum. This can greatly reduced fit time compared to creating pipeline classes for each estimator and pipeline and fitting them one at a time in an Scikit-learn `sklearn.model_selection.GridSearch` class.

Preprocessing can be done before making any evaluation, and several evaluations can be made on the pre-made folds. Current implementation relies on a randomized grid search, so parameter grids must be specified as SciPy distributions (or a class that accepts a `rvs` method).

Parameters

- **scorer** (*function*) – a scoring function that follows the Scikit-learn API:

```
score = scorer(estimator, y_true, y_pred)
```

  A user defines scoring function, `score = f(y_true, y_pred)` can be made into a scorer by calling on the ML-Ensemble implementation of Scikit-learn's `make_scorer`. NOTE: do **not** use Scikit-learn's `make_scorer` if the Evaluator is to be pickled.

```
from mlens.metrics import make_scorer
scorer = make_scorer(scoring_function, **kwargs)
```

- **error_score** (*int, optional*) – score to assign when fitting an estimator fails. If `None`, the evaluator will raise an error.
- **cv** (*int or obj (default = 2)*) – cross validation folds to use. Either pass a `KFold` class that obeys the Scikit-learn API.
- **metrics** (*list, optional*) – list of aggregation metrics to calculate on scores. Default is mean and standard deviation.
- **shuffle** (*bool (default = True)*) – whether to shuffle input data before creating cv folds.
- **random_state** (*int, optional*) – seed for creating folds (if shuffled) and parameter draws

- **array_check** (*int (default = 2)*) – level of strictness in checking input arrays.
  - `array_check = 0` will not check `X` or `y`
  - `array_check = 1` will check `X` and `y` for inconsistencies and warn when format looks suspicious, but retain original format.
  - `array_check = 2` will impose Scikit-learn array checks, which converts `X` and `y` to numpy arrays and raises an error if conversion fails.
- **n_jobs** (*int (default = -1)*) – number of CPU cores to use.
- **verbose** (*bool or int (default = False)*) – level of printed messages.

summary
dict – Summary output that shows data for best mean test scores, such as test and train scores, std, fit times, and params.

cv_results
dict – a nested `dict` of data from each fit. Includes mean and std of test and train scores and fit times, as well as param draw index and parameters.

evaluate(*X*, *y*, *estimators*, *param_dicts*, *n_iter=2*)
Evaluate set of estimators.

Function for evaluating a set of estimators using cross validation. Similar to a randomized grid search, but applies the grid search to all specified preprocessing pipelines.

Parameters

- **X** (*array-like, shape=[n_samples, n_features]*) – input data to preprocess and create folds from.
- **y** (*array-like, shape=[n_samples,]*) – training labels.
- **estimators** (*list or dict*) – set of estimators to use. If no preprocessing is desired or if only on preprocessing pipeline should apply to all, pass a list of estimators. The list can contain elements of named tuples (i.e. `('my_name', my_est)`).

  If different estimators should be mapped to preprocessing cases, a dictionary that maps estimators to each case should be passed: `{'case_a': list_of_est, ...}`.
- **param_dicts** (*dict*) – parameter distribution mapping for estimators. Current implementation only supports randomized grid search. Passed distribution object must have an `rvs` method. See `Scipy.stats` for details.

  There is quite some flexibility in specifying `param_dicts`. If there is no preprocessing, or if all estimators are fitted on all preprocessing cases, the `param_dict` should have keys matching the names of the estimators.

```
estimators = [('name', est), est]

param_dicts = {'name': {'param-1': some_distribution},
               'est': {'param-1': some_distribution}
              }
```

It is possible to specify different distributions for some or all preprocessing cases:

```
preprocessing = {'case-1': transformer_list,
                 'case-2': transformer_list}

estimators = [('name', est), est]
```

```
param_dicts = {'name':
                    {'param-1': some_distribution},
               ('case-1', 'est'):
                    {'param-1': some_distribution}
               ('case-2', 'est'):
                    {'param-1': some_distribution,
                     'param-2': some_distribution}
              }
```

If estimators are mapped on a per-preprocessing case basis as a dictionary, `param_dict` must have key entries of the form `(case_name, est_name)`.

- **n_iter** (*int*) – number of parameter draws to evaluate.

Returns **self** – class instance with stored estimator evaluation results.

Return type instance

fit (*X*, *y*, *estimators*, *param_dicts*, *n_iter=2*, *preprocessing=None*)

Fit the Evaluator to given data, estimators and preprocessing.

Utility function that calls `preprocess` and `evaluate`. The following is equivalent:

```
# Explicitly calling preprocess and evaluate
evaluator.preprocess(X, y, preprocessing)
evaluator.evaluate(X, y, estimators, param_dicts, n_iter)

# Calling fit
evaluator.fit(X, y, estimators, param_dicts, n_iter, preprocessing)
```

Parameters

- **X** (*array-like, shape=[n_samples, n_features]*) – input data to preprocess and create folds from.
- **y** (*array-like, shape=[n_samples,]*) – training labels.
- **estimators** (*list or dict*) – set of estimators to use. If no preprocessing is desired or if only on preprocessing pipeline should apply to all, pass a list of estimators. The list can contain elements of named tuples (i.e. `('my_name', my_est)`).

  If different estimators should be mapped to preprocessing cases, a dictionary that maps estimators to each case should be passed: `{'case_a': list_of_est, ...}`.
- **param_dicts** (*dict*) – parameter distribution mapping for estimators. Current implementation only supports randomized grid search. Passed distribution object must have an `rvs` method. See `Scipy.stats` for details.

  There is quite some flexibility in specifying `param_dicts`. If there is no preprocessing, or if all estimators are fitted on all preprocessing cases, the `param_dict` should have keys matching the names of the estimators.

```
estimators = [('name', est), est]

param_dicts = {'name': {'param-1': some_distribution},
               'est': {'param-1': some_distribution}
              }
```

It is possible to specify different distributions for some or all preprocessing cases:

```
preprocessing = {'case-1': transformer_list,
                 'case-2': transformer_list}

estimators = [('name', est), est]

param_dicts = {'name':
                   {'param-1': some_distribution},
               ('case-1', 'est'):
                   {'param-1': some_distribution}
               ('case-2', 'est'):
                   {'param-1': some_distribution,
                    'param-2': some_distribution}
              }
```

If estimators are mapped on a per-preprocessing case basis as a dictionary, `param_dict` must have key entries of the form `(case_name, est_name)`.

- **n_iter** (*int*) – number of parameter draws to evaluate.
- **preprocessing** (*dict, optional*) – preprocessing cases to consider. Pass a dictionary mapping a case name to a preprocessing pipeline.

```
preprocessing = {'case_name': transformer_list,}
```

Returns **self** – class instance with stored estimator evaluation results.

Return type instance

initialize(*X*, *y*)
Set up `ParallelEvaluation` job manager.

preprocess(*X*, *y*, *preprocessing=None*)
Preprocess folds.

Method for preprocessing data separately from the evaluation method. Helpful if preprocessing is costly relative to estimator fitting and several `evaluate` calls might be desired.

Parameters

- **X** (*array-like, shape=[n_samples, n_features]*) – input data to preprocess and create folds from.
- **y** (*array-like, shape=[n_samples,]*) – training labels.
- **preprocessing** (*list or dict, optional*) – preprocessing cases to consider. Pass a dictionary mapping a case name to a preprocessing pipeline.

```
preprocessing = {'case_name': transformer_list,}
```

Returns **self** – class instance with stored estimator evaluation results.

Return type instance

terminate()
Terminate evaluation job.

mlens.parallel package

Submodules

mlens.parallel.blend module

ML-ENSEMBLE

author Sebastian Flennerhag

copyright 2017

licence MIT

Estimation engine for parallel preprocessing of blend layer.

class mlens.parallel.blend.**Blender**(*job*, *layer*)
Bases: *mlens.parallel.estimation.BaseEstimator*

Blended fit sub-process class.

Class for fitting a Layer using Blending.

run(*parallel*)
Execute stacking.

mlens.parallel.blend.**transform**(*inst*, *X*, *P*, *parallel*)
Predict X.

Since a blend ensemble does not use folds, transform coincides with predict, except that the prediction in fitting is only for a subset of X.

mlens.parallel.estimation module

ML-ENSEMBLE

author Sebastian Flennerhag

copyright 2017

licence MIT

Base class for estimation.

class mlens.parallel.estimation.**BaseEstimator**(*layer*)
Bases: object

Base class for estimating a layer in parallel.

Estimation class to be used as based for a layer estimation engined that is callable by the ParallelProcess job manager.

A subclass must implement a constructor that accepts the following args:

- job : the Job instance containing relevant data
- layer: the Layer instance to estimate
- n: the position in the LayerContainer stack of the layer

as well as a run method that accepts a Parallel instance.

Parameters **layer** (Layer) – layer to be estimated.

run (*parallel*)
Method for executing estimation.

Default method relies on the default constructor. Both can be replaced if desired.

Parameters **parallel** (*object*) – Parallel instance.

mlens.parallel.evaluation module

ML-ENSEMBLE

author Sebastian Flennerhag

copyright 2017

licence MIT

Cross-validation jobs for an Evaluator instance.

class mlens.parallel.evaluation.**Evaluation**(*evaluator*)
Bases: object

Evaluation engine.

Run a job for an Evaluator instance.

Parameters **evaluator** (Evaluator) – Evaluator instance to run job for.

evaluate(*parallel*, *X*, *y*, *dir*)
cross-validation of estimators.

Parameters

- **parallel** (joblib.Parallel) – The instance to use for parallel fitting.
- **X** (*array-like of shape [n_samples, n_features]*) – Training set to use for estimation. Can be memmaped.
- **y** (*array-like of shape [n_samples,]*) – labels for estimation. Can be memmaped.
- **dir** (*str*) – directory of cache to dump fitted transformers before assembly.

preprocess(*parallel*, *X*, *y*, *dir*)
Fit preprocessing pipelines.

Fit all preprocessing pipelines in parallel and store as a preprocessing_ attribute on the Evaluator.

Parameters

- **parallel** (joblib.Parallel) – The instance to use for parallel fitting.
- **X** (*array-like of shape [n_samples, n_features]*) – Training set to use for estimation. Can be memmaped.
- **y** (*array-like of shape [n_samples,]*) – labels for estimation. Can be memmaped.
- **dir** (*directory of cache to dump fitted transformers before assembly.*) –

mlens.parallel.evaluation.**fit_score**(*case*, *tr_list*, *est_name*, *est*, *params*, *x*, *y*, *idx*, *scorer*, *error_score*)
Wrapper around fit function to determine how to handle exceptions.

mlens.parallel.manager module

ML-Ensemble

author Sebastian Flennerhag

copyright 2017

licence MIT

Parallel processing job managers.

class mlens.parallel.manager.**BaseProcessor**(*caller*)
Bases: object

Parallel processing base class.

Base class for parallel processing engines.

caller

initialize(*job*, *X*, *y=None*, *dir=None*)
Create a job instance for estimation.

job

terminate()
Remove temporary folder and all cache data.

class mlens.parallel.manager.**Job**(*job*)
Bases: object

Container class for holding job data.

See also:

ParallelProcessing, *ParallelEvaluation*

dir

job

predict_in

predict_out

tmp

update()
Shift output array to input array.

y

class mlens.parallel.manager.**ParallelEvaluation**(*caller*)
Bases: *mlens.parallel.manager.BaseProcessor*

Parallel cross-validation engine.

Parameters **caller** (Evaluator) – The Evaluator that instantiated the processor.

process(*attr*)
Fit all layers in the attached LayerContainer.

class mlens.parallel.manager.**ParallelProcessing**(*caller*)
Bases: *mlens.parallel.manager.BaseProcessor*

Parallel processing engine.

Engine for running ensemble estimation.

Parameters **layers** (*mlens.ensemble.base.LayerContainer*) – The LayerContainer that instantiated the processor.

get_preds(*dtype=None*, *order='C'*)
Return prediction matrix.

Parameters

- **dtype** (*numpy dtype object, optional*) – data type to return
- **order** (*str (default = 'C')*) – data order. See numpy.asarray for details.

process()
Fit all layers in the attached LayerContainer.

mlens.parallel.manager.**dump_array**(*array*, *name*, *dir*)
Dump array for memmapping.

mlens.parallel.single_run module

ML-ENSEMBLE

author Sebastian Flennerhag

copyright 2017

licence MIT

Estimation engine for parallel preprocessing of estimators in a single run, such as when fitting a final layer (meta estimator) that does not require propagating predictions.

class mlens.parallel.single_run.**SingleRun**(*job*, *layer*)
Bases: *mlens.parallel.estimation.BaseEstimator*

Single run fit sub-process class.

Class for fitting a estimators in a layer without any sub-fits.

run(*parallel*)
Execute blending.

mlens.parallel.stack module

ML-ENSEMBLE

author Sebastian Flennerhag

copyright 2017

licence MIT

Estimation engine for parallel preprocessing of stacked layer.

class mlens.parallel.stack.**Stacker**(*job*, *layer*)
Bases: *mlens.parallel.estimation.BaseEstimator*

Stacked fit sub-process class.

Class for fitting a Layer using Stacking.

run (*parallel*)
Execute stacking.

mlens.parallel.subset module

ML-ENSEMBLE

author Sebastian Flennerhag

copyright 2017

licence MIT

Estimation engine for parallel preprocessing of subsemble layer.

class mlens.parallel.subset.**SubStacker**(*job*, *layer*)
Bases: *mlens.parallel.estimation.BaseEstimator*

Stacked subset fit sub-process class.

Class for fitting a Layer using Subsemble.

run (*parallel*)
Execute subsembling

Module contents

ML-ENSEMBLE

author Sebastian Flennerhag

copyright 2017

licence MIT

class mlens.parallel.**ParallelProcessing**(*caller*)
Bases: *mlens.parallel.manager.BaseProcessor*

Parallel processing engine.

Engine for running ensemble estimation.

Parameters **layers** (*mlens.ensemble.base.LayerContainer*) – The LayerContainer that instantiated the processor.

get_preds (*dtype=None*, *order='C'*)
Return prediction matrix.

Parameters

- **dtype** (*numpy dtype object, optional*) – data type to return
- **order** (*str (default = 'C')*) – data order. See numpy.asarray for details.

process()
Fit all layers in the attached LayerContainer.

class mlens.parallel.**ParallelEvaluation**(*caller*)
Bases: *mlens.parallel.manager.BaseProcessor*

Parallel cross-validation engine.

Parameters **caller** (Evaluator) – The Evaluator that instantiated the processor.

process (*attr*)
Fit all layers in the attached `LayerContainer`.

class `mlens.parallel.`**Stacker** (*job*, *layer*)
Bases: *mlens.parallel.estimation.BaseEstimator*

Stacked fit sub-process class.

Class for fitting a Layer using Stacking.

run (*parallel*)
Execute stacking.

class `mlens.parallel.`**Blender** (*job*, *layer*)
Bases: *mlens.parallel.estimation.BaseEstimator*

Blended fit sub-process class.

Class for fitting a Layer using Blending.

run (*parallel*)
Execute stacking.

class `mlens.parallel.`**SubStacker** (*job*, *layer*)
Bases: *mlens.parallel.estimation.BaseEstimator*

Stacked subset fit sub-process class.

Class for fitting a Layer using Subsemble.

run (*parallel*)
Execute subsembling

class `mlens.parallel.`**SingleRun** (*job*, *layer*)
Bases: *mlens.parallel.estimation.BaseEstimator*

Single run fit sub-process class.

Class for fitting a estimators in a layer without any sub-fits.

run (*parallel*)
Execute blending.

class `mlens.parallel.`**Evaluation** (*evaluator*)
Bases: `object`

Evaluation engine.

Run a job for an `Evaluator` instance.

Parameters **evaluator** (`Evaluator`) – Evaluator instance to run job for.

evaluate (*parallel*, *X*, *y*, *dir*)
cross-validation of estimators.

Parameters

- **parallel** (`joblib.Parallel`) – The instance to use for parallel fitting.
- **X** (*array-like of shape [n_samples, n_features]*) – Training set to use for estimation. Can be memmaped.
- **y** (*array-like of shape [n_samples,]*) – labels for estimation. Can be memmaped.
- **dir** (*str*) – directory of cache to dump fitted transformers before assembly.

preprocess (*parallel*, *X*, *y*, *dir*)
Fit preprocessing pipelines.

Fit all preprocessing pipelines in parallel and store as a `preprocessing_` attribute on the `Evaluator`.

Parameters

- **parallel** (`joblib.Parallel`) – The instance to use for parallel fitting.
- **X** (*array-like of shape [n_samples, n_features]*) – Training set to use for estimation. Can be memmaped.
- **y** (*array-like of shape [n_samples,]*) – labels for estimation. Can be memmaped.
- **dir** (*directory of cache to dump fitted transformers before assembly.*) –

class `mlens.parallel.`**BaseEstimator** (*layer*)
Bases: `object`

Base class for estimating a layer in parallel.

Estimation class to be used as based for a layer estimation engined that is callable by the `ParallelProcess` job manager.

A subclass must implement a constructor that accepts the following args:

- `job` : the `Job` instance containing relevant data
- `layer`: the `Layer` instance to estimate
- `n`: the position in the `LayerContainer` stack of the layer

as well as a `run` method that accepts a `Parallel` instance.

Parameters **layer** (`Layer`) – layer to be estimated.

run (*parallel*)
Method for executing estimation.

Default method relies on the default constructor. Both can be replaced if desired.

Parameters **parallel** (*object*) – `Parallel` instance.

mlens.preprocessing package

Submodules

mlens.preprocessing.ensemble_transformer module

ML-ENSEMBLE

author Sebastian Flennerhag

copyright 2017

licence MIT

Ensemble transformer class. Fully integrable with Scikit-learn.

class mlens.preprocessing.ensemble_transformer.**EnsembleTransformer**(*shuffle=False*, *random_state=None*, *scorer=None*, *raise_on_exception=True*, *array_check=2*, *verbose=False*, *n_jobs=1*, *layers=None*, *backend=None*, *sample_dim=10*)

Bases: *mlens.ensemble.base.BaseEnsemble*

Ensemble Transformer class.

The Ensemble class allows users to build layers of an ensemble through a transformer API. The transformer is closely related to `SequentialEnsemble`, in that any accepted type of layer can be added. The transformer differs fundamentally in one significant aspect: when fitted, it will store a random sample of the training set together with the training dimensions, and if in a call to `transform`, the data to be transformed correspodns to the training set, the transformer will recreate the prediction matrix from the `fit` call. In contrast, a fitted ensemble will only use the base learners fitted on the full dataset, and as such predicting the training set will not reproduce the predictions from the `fit` call.

The *EnsembleTransformer* is a powerful tool to use as a preprocessing pipeline in an `Evaluator` instance, as it would faithfully recreate the prediction matrix a potential meta learner would face. Hence, a user can 'preprocess' the training data with the *EnsembleTransformer* to generate k-fold base learner predictions, and then fit different meta learners (or higher-order layers) in a call to `evaluate`.

See also:

`SequentialEnsemble`, `Evaluator`

Parameters

- **shuffle** (*bool (default = True)*) – whether to shuffle data before generating folds.
- **random_state** (*int (default = None)*) – random seed if shuffling inputs.
- **scorer** (*object (default = None)*) – scoring function. If a function is provided, base estimators will be scored on the training set assembled for fitting the meta estimator. Since those predictions are out-of-sample, the scores represent valid test scores. The scorer should be a function that accepts an array of true values and an array of predictions: `score = f(y_true, y_pred)`.
- **raise_on_exception** (*bool (default = True)*) – whether to issue warnings on soft exceptions or raise error. Examples include lack of layers, bad inputs, and failed fit of an estimator in a layer. If set to `False`, warnings are issued instead but estimation continues unless exception is fatal. Note that this can result in unexpected behavior unless the exception is anticipated.
- **sample_dim**(*int (default = 10)*) – dimensionality of training set to sample. During a call to *fit*, a random sample of size [sample_dim, sample_dim] will be sampled from the training data, along with the dimensions of the training data. If in a call to `transform`, sampling the same indices on the array to transform gives the same sample matrix, the

transformer will reproduce the predictions from the call to `fit`, as opposed to using the base learners fitted on the full training data.

- **array_check** (*int (default = 2)*) – level of strictness in checking input arrays.
  - `array_check = 0` will not check `X` or `y`
  - `array_check = 1` will check `X` and `y` for inconsistencies and warn when format looks suspicious, but retain original format.
  - `array_check = 2` will impose Scikit-learn array checks, which converts `X` and `y` to numpy arrays and raises an error if conversion fails.
- **verbose** (*int or bool (default = False)*) – level of verbosity.
  - `verbose = 0` silent (same as `verbose = False`)
  - `verbose = 1` messages at start and finish (same as `verbose = True`)
  - `verbose = 2` messages for each layer

  If `verbose >= 50` prints to `sys.stdout`, else `sys.stderr`. For verbosity in the layers themselves, use `fit_params`.
- **n_jobs** (*int (default = 1)*) – number of CPU cores to use for fitting and prediction.

scores_

dict – if `scorer` was passed to instance, `scores_` contains dictionary with cross-validated scores assembled during `fit` call. The fold structure used for scoring is determined by `folds`.

Examples

```
>>> from mlens.preprocessing import EnsembleTransformer
>>> from mlens.model_selection import Evaluator
>>> from mlens.metrics.metrics import rmse
>>> from sklearn.datasets import load_boston
>>> from sklearn.linear_model import Lasso
>>> from sklearn.svm import SVR
>>> from scipy.stats import uniform
>>> from pandas import DataFrame
>>>
>>> X, y = load_boston(True)
>>>
>>> ensemble = EnsembleTransformer()
>>>
>>> ensemble.add('stack', [SVR(), Lasso()])
>>>
>>> evl = Evaluator(scorer=rmse, random_state=10)
>>>
>>> evl.preprocess(X, y, [('scale', ensemble)])
>>>
>>> draws = {(None, 'svr'): {'C': uniform(10,  100)},
...          (None, 'lasso'): {'alpha': uniform(0.01, 0.1)}}
>>>
>>> evl.evaluate(X, y, [SVR(), Lasso()], draws, n_iter=10)
>>>
>>> DataFrame(evl.summary)
       fit_time_mean  fit_time_std  test_score_mean  test_score_std  \
lasso       0.000818      0.000362         7.514181        0.827578
```

```
svr          0.009790      0.000596        10.949149        0.577554
       train_score_mean  train_score_std                      params
lasso          6.228287         0.949872  {'alpha': 0.0871320643267}
svr            5.794856         1.348409        {'C': 12.0751949359}
```

add(*cls*, *estimators*, *preprocessing=None*, ***kwargs*)

Add layer to ensemble transformer.

Parameters

- **cls** (*str*) – layer class. Accepted types are:
  - 'blend' : blend ensemble
  - 'subset' : subsemble
  - 'stack' : super learner
- **estimators** (*dict of lists or list or instance*) – estimators constituting the layer. If preprocessing is none and the layer is meant to be the meta estimator, it is permissible to pass a single instantiated estimator. If `preprocessing` is `None` or `list`, `estimators` should be a `list`. The list can either contain estimator instances, named tuples of estimator instances, or a combination of both.

```
option_1 = [estimator_1, estimator_2]
option_2 = [("est-1", estimator_1), ("est-2", estimator_2)]
option_3 = [estimator_1, ("est-2", estimator_2)]
```

If different preprocessing pipelines are desired, a dictionary that maps estimators to preprocessing pipelines must be passed. The names of the estimator dictionary must correspond to the names of the estimator dictionary.

```
preprocessing_cases = {"case-1": [trans_1, trans_2],
                       "case-2": [alt_trans_1, alt_trans_2]}

estimators = {"case-1": [est_a, est_b],
              "case-2": [est_c, est_d]}
```

The lists for each dictionary entry can be any of `option_1`, `option_2` and `option_3`.

- **preprocessing** (*dict of lists or list, optional (default = None)*) – preprocessing pipelines for given layer. If the same preprocessing applies to all estimators, `preprocessing` should be a list of transformer instances. The list can contain the instances directly, named tuples of transformers, or a combination of both.

```
option_1 = [transformer_1, transformer_2]
option_2 = [("trans-1", transformer_1),
            ("trans-2", transformer_2)]
option_3 = [transformer_1, ("trans-2", transformer_2)]
```

If different preprocessing pipelines are desired, a dictionary that maps preprocessing pipelines must be passed. The names of the preprocessing dictionary must correspond to the names of the estimator dictionary.

```
preprocessing_cases = {"case-1": [trans_1, trans_2],
                       "case-2": [alt_trans_1, alt_trans_2]}

estimators = {"case-1": [est_a, est_b],
              "case-2": [est_c, est_d]}
```

The lists for each dictionary entry can be any of `option_1`, `option_2` and `option_3`.

- ****kwargs** (*optional*) – optional keyword arguments to instantiate layer with. See respective ensemble for further details.

Returns **self** – ensemble instance with layer instantiated.

Return type instance

fit (*X*, *y=None*)
Fit the transformer.

Same as the fit method on an ensemble, except that a sample of X is stored for future comparison.

predict (*X*)
Generate predictions for X. Same as `transform`.

transform (*X*, *y=None*)
Transform input X into a prediction matrix Z.

If X is the training set, the transformer will reproduce the Z from the call to `fit`. If X is another data set, Z will be produced using base learners fitted on the full training data (equivalent to calling `predict` on an ensemble.)

mlens.preprocessing.preprocess module

ML-ENSEMBLE

author Sebastian Flennerhag

copyright 2017

licence MIT

class `mlens.preprocessing.preprocess.`**Shift** (*s*)
Bases: `mlens.externals.sklearn.base.BaseEstimator`, `mlens.externals.sklearn.base.TransformerMixin`

Lag operator.

Shift an input array X with s steps, i.e. for some time series $\mathbf{X} = (X_t, X_{t-1}, ..., X_0)$,

$$L^s\mathbf{X} = (X_{t-s}, X_{t-1-s}, ..., X_{s-s})$$

Parameters **s** (*int*) – number of lags to generate

Examples

```
>>> import numpy as np
>>> from mlens.preprocessing import Shift
>>> X = np.arange(10)
>>> L = Shift(2)
>>> Z = L.fit_transform(X)
>>> print("X : {}".format(X[2:]))
>>> print("Z : {}".format(Z))
X : [2 3 4 5 6 7 8 9]
Z : [0 1 2 3 4 5 6 7]
```

fit (*X*, *y=None*)
Pass through for compatability.

transform(*X*)
Return lagged dataset.

class mlens.preprocessing.preprocess.**Subset**(*subset=None*)
Bases: mlens.externals.sklearn.base.BaseEstimator, mlens.externals.sklearn.base.TransformerMixin

Select a subset of features.

The Subset class acts as a transformer that reduces the feature set to a subset specified by the user.

Parameters **subset** (*list*) – list of columns indexes to select subset with. Indexes can either be of type str if data accepts slicing on a list of strings, otherwise the list should be of type int.

fit(*X*, *y=None*)
Learn what format the data is stored in.

Parameters

- **X** (*array-like of shape = [n_samples, n_features]*) – The whose type will be inferred.
- **y** (*array-like of shape = [n_samples, n_features]*) – pass-through for Scikit-learn pipeline compatibility.

transform(*X*, *y=None*, *copy=False*)
Return specified subset of X.

Parameters

- **X** (*array-like of shape = [n_samples, n_features]*) – The whose type will be inferred.
- **y** (*array-like of shape = [n_samples, n_features]*) – pass-through for Scikit-learn pipeline compatibility.
- **copy** (*bool (default = None)*) – whether to copy X before transforming.

Module contents

author Sebastian Flennerhag

copyright 2017

licence MIT

class mlens.preprocessing.**Subset**(*subset=None*)
Bases: mlens.externals.sklearn.base.BaseEstimator, mlens.externals.sklearn.base.TransformerMixin

Select a subset of features.

The Subset class acts as a transformer that reduces the feature set to a subset specified by the user.

Parameters **subset** (*list*) – list of columns indexes to select subset with. Indexes can either be of type str if data accepts slicing on a list of strings, otherwise the list should be of type int.

fit(*X*, *y=None*)
Learn what format the data is stored in.

Parameters

- **X** (*array-like of shape = [n_samples, n_features]*) – The whose type will be inferred.

- **y** (*array-like of shape = [n_samples, n_features]*) – pass-through for Scikit-learn pipeline compatibility.

transform(*X*, *y=None*, *copy=False*)
Return specified subset of X.

Parameters

- **X** (*array-like of shape = [n_samples, n_features]*) – The whose type will be inferred.
- **y** (*array-like of shape = [n_samples, n_features]*) – pass-through for Scikit-learn pipeline compatibility.
- **copy** (*bool (default = None)*) – whether to copy X before transforming.

class mlens.preprocessing.**Shift**(*s*)
Bases: mlens.externals.sklearn.base.BaseEstimator, mlens.externals.sklearn.base.TransformerMixin

Lag operator.

Shift an input array X with s steps, i.e. for some time series $\mathbf{X} = (X_t, X_{t-1}, ..., X_0)$,

$$L^s\mathbf{X} = (X_{t-s}, X_{t-1-s}, ..., X_{s-s})$$

Parameters **s** (*int*) – number of lags to generate

Examples

```
>>> import numpy as np
>>> from mlens.preprocessing import Shift
>>> X = np.arange(10)
>>> L = Shift(2)
>>> Z = L.fit_transform(X)
>>> print("X : {}".format(X[2:]))
>>> print("Z : {}".format(Z))
X : [2 3 4 5 6 7 8 9]
Z : [0 1 2 3 4 5 6 7]
```

fit(*X*, *y=None*)
Pass through for compatability.

transform(*X*)
Return lagged dataset.

class mlens.preprocessing.**EnsembleTransformer**(*shuffle=False*, *random_state=None*, *scorer=None*, *raise_on_exception=True*, *array_check=2*, *verbose=False*, *n_jobs=1*, *layers=None*, *backend=None*, *sample_dim=10*)
Bases: *mlens.ensemble.base.BaseEnsemble*

Ensemble Transformer class.

The Ensemble class allows users to build layers of an ensemble through a transformer API. The transformer is closely related to SequentialEnsemble, in that any accepted type of layer can be added. The transformer differs fundamentally in one significant aspect: when fitted, it will store a random sample of the training set together with the training dimensions, and if in a call to transform, the data to be transformed correspodns to the training set, the transformer will recreate the prediction matrix from the fit call. In contrast, a fitted

ensemble will only use the base learners fitted on the full dataset, and as such predicting the training set will not reproduce the predictions from the `fit` call.

The *EnsembleTransformer* is a powerful tool to use as a preprocessing pipeline in an `Evaluator` instance, as it would faithfully recreate the prediction matrix a potential meta learner would face. Hence, a user can 'preprocess' the training data with the *EnsembleTransformer* to generate k-fold base learner predictions, and then fit different meta learners (or higher-order layers) in a call to `evaluate`.

See also:

`SequentialEnsemble`, `Evaluator`

Parameters

- **shuffle** (*bool (default = True)*) – whether to shuffle data before generating folds.
- **random_state** (*int (default = None)*) – random seed if shuffling inputs.
- **scorer** (*object (default = None)*) – scoring function. If a function is provided, base estimators will be scored on the training set assembled for fitting the meta estimator. Since those predictions are out-of-sample, the scores represent valid test scores. The scorer should be a function that accepts an array of true values and an array of predictions: `score = f(y_true, y_pred)`.
- **raise_on_exception** (*bool (default = True)*) – whether to issue warnings on soft exceptions or raise error. Examples include lack of layers, bad inputs, and failed fit of an estimator in a layer. If set to `False`, warnings are issued instead but estimation continues unless exception is fatal. Note that this can result in unexpected behavior unless the exception is anticipated.
- **sample_dim** (*int (default = 10)*) – dimensionality of training set to sample. During a call to *fit*, a random sample of size [sample_dim, sample_dim] will be sampled from the training data, along with the dimensions of the training data. If in a call to `transform`, sampling the same indices on the array to transform gives the same sample matrix, the transformer will reproduce the predictions from the call to `fit`, as opposed to using the base learners fitted on the full training data.
- **array_check** (*int (default = 2)*) – level of strictness in checking input arrays.
  - `array_check = 0` will not check `X` or `y`
  - `array_check = 1` will check `X` and `y` for inconsistencies and warn when format looks suspicious, but retain original format.
  - `array_check = 2` will impose Scikit-learn array checks, which converts `X` and `y` to numpy arrays and raises an error if conversion fails.
- **verbose** (*int or bool (default = False)*) – level of verbosity.
  - `verbose = 0` silent (same as `verbose = False`)
  - `verbose = 1` messages at start and finish (same as `verbose = True`)
  - `verbose = 2` messages for each layer

  If `verbose >= 50` prints to `sys.stdout`, else `sys.stderr`. For verbosity in the layers themselves, use `fit_params`.
- **n_jobs** (*int (default = 1)*) – number of CPU cores to use for fitting and prediction.

scores_
dict – if `scorer` was passed to instance, `scores_` contains dictionary with cross-validated scores assembled during `fit` call. The fold structure used for scoring is determined by `folds`.

Examples

```
>>> from mlens.preprocessing import EnsembleTransformer
>>> from mlens.model_selection import Evaluator
>>> from mlens.metrics.metrics import rmse
>>> from sklearn.datasets import load_boston
>>> from sklearn.linear_model import Lasso
>>> from sklearn.svm import SVR
>>> from scipy.stats import uniform
>>> from pandas import DataFrame
>>>
>>> X, y = load_boston(True)
>>>
>>> ensemble = EnsembleTransformer()
>>>
>>> ensemble.add('stack', [SVR(), Lasso()])
>>>
>>> evl = Evaluator(scorer=rmse, random_state=10)
>>>
>>> evl.preprocess(X, y, [('scale', ensemble)])
>>>
>>> draws = {(None, 'svr'): {'C': uniform(10,  100)},
...          (None, 'lasso'): {'alpha': uniform(0.01, 0.1)}}
>>>
>>> evl.evaluate(X, y, [SVR(), Lasso()], draws, n_iter=10)
>>>
>>> DataFrame(evl.summary)
       fit_time_mean  fit_time_std  test_score_mean  test_score_std  \
lasso       0.000818      0.000362         7.514181        0.827578
svr         0.009790      0.000596        10.949149        0.577554
       train_score_mean  train_score_std                      params
lasso          6.228287         0.949872  {'alpha': 0.0871320643267}
svr            5.794856         1.348409        {'C': 12.0751949359}
```

add(*cls*, *estimators*, *preprocessing=None*, ***kwargs*)
Add layer to ensemble transformer.

Parameters

- **cls** (*str*) – layer class. Accepted types are:
  - 'blend' : blend ensemble
  - 'subset' : subsemble
  - 'stack' : super learner
- **estimators** (*dict of lists or list or instance*) – estimators constituting the layer. If preprocessing is none and the layer is meant to be the meta estimator, it is permissible to pass a single instantiated estimator. If `preprocessing` is `None` or `list`, `estimators` should be a `list`. The list can either contain estimator instances, named tuples of estimator instances, or a combination of both.

```
option_1 = [estimator_1, estimator_2]
option_2 = [("est-1", estimator_1), ("est-2", estimator_2)]
option_3 = [estimator_1, ("est-2", estimator_2)]
```

If different preprocessing pipelines are desired, a dictionary that maps estimators to preprocessing pipelines must be passed. The names of the estimator dictionary must correspond to the names of the estimator dictionary.

```
preprocessing_cases = {"case-1": [trans_1, trans_2],
                       "case-2": [alt_trans_1, alt_trans_2]}

estimators = {"case-1": [est_a, est_b],
              "case-2": [est_c, est_d]}
```

The lists for each dictionary entry can be any of `option_1`, `option_2` and `option_3`.

- **preprocessing** (*dict of lists or list, optional (default = None)*) – preprocessing pipelines for given layer. If the same preprocessing applies to all estimators, `preprocessing` should be a list of transformer instances. The list can contain the instances directly, named tuples of transformers, or a combination of both.

```
option_1 = [transformer_1, transformer_2]
option_2 = [("trans-1", transformer_1),
            ("trans-2", transformer_2)]
option_3 = [transformer_1, ("trans-2", transformer_2)]
```

If different preprocessing pipelines are desired, a dictionary that maps preprocessing pipelines must be passed. The names of the preprocessing dictionary must correspond to the names of the estimator dictionary.

```
preprocessing_cases = {"case-1": [trans_1, trans_2],
                       "case-2": [alt_trans_1, alt_trans_2]}

estimators = {"case-1": [est_a, est_b],
              "case-2": [est_c, est_d]}
```

The lists for each dictionary entry can be any of `option_1`, `option_2` and `option_3`.

- ****kwargs** (*optional*) – optional keyword arguments to instantiate layer with. See respective ensemble for further details.

Returns **self** – ensemble instance with layer instantiated.

Return type instance

fit (*X*, *y=None*)
Fit the transformer.

Same as the fit method on an ensemble, except that a sample of X is stored for future comparison.

predict (*X*)
Generate predictions for X. Same as `transform`.

transform (*X*, *y=None*)
Transform input X into a prediction matrix Z.

If X is the training set, the transformer will reproduce the Z from the call to `fit`. If X is another data set, Z will be produced using base learners fitted on the full training data (equivalent to calling `predict` on an ensemble.)

mlens.utils package

Submodules

mlens.utils.checks module

ML-ENSEMBLE

author Sebastian Flennerhag

copyright 2017

licence MIT

Controls that an estimator was built as expected.

mlens.utils.checks.**assert_correct_format**(*estimators*, *preprocessing*)
Initial check to assert layer can be constructed.

mlens.utils.checks.**assert_valid_estimator**(*instance*)
Assert that an instance has a `get_params` and `fit` method.

mlens.utils.checks.**check_ensemble_build**(*inst*, *attr='layers'*)
Check that layers have been instantiated.

mlens.utils.checks.**check_initialized**(*inst*)
Check if a ParallelProcessing instance is initialized properly.

mlens.utils.checks.**check_is_fitted**(*estimator*, *attr*)
Check that ensemble has been fitted.

Parameters

- **estimator** (*estimator instance*) – ensemble instance to check.
- **attr** (*str*) – attribute to assert existence of.

mlens.utils.dummy module

ML-ENSEMBLE

author Sebastian Flennerhag

copyright 2017

license MIT

Collection of dummy estimator classes, Mixins to build transparent layers for unit testing.

Also contains pre-made Layer, LayerContainers and data generation functions for unit testing.

class mlens.utils.dummy.**Cache**(*X*, *y*, *data*)
Bases: `object`

Object for controlling caching.

layer_est(*layer*, *attr*)
Test the estimation routine for a layer.

store_X_y(*X*, *y*, *as_csv=False*)
Save X and y to file in temporary directory.

terminate()
Remove temporary items in directory during tests.

class mlens.utils.dummy.**Data**(*cls*, *proba*, *preprocessing*, **args*, ***kwargs*)
Bases: object

Class for getting data.

get_data(*shape*, *m*)
Generate X and y data with X.

Parameters

- **shape** (*tuple*) – shape of data to be generated
- **m** (*int*) – length of step function for y

Returns

- **train** (*ndarray*) – generated as a sequence of reshaped to (LEN, WIDTH)
- **labels** (*ndarray*) – generated as a step-function with a step every m. As such, each prediction fold during cross-validation have a unique level value.

ground_truth(*X*, *y*, *subsets=1*, *verbose=False*)
Set up an experiment ground truth.

Returns

- **F** (*ndarray*) – Full prediction array (train errors)
- **P** (*ndarray*) – Folded prediction array (test errors)

Raises *AssertionError :* – Raises assertion error if any weight vectors overlap or any predictions (as measured by columns in F and P) are judged to be equal.

class mlens.utils.dummy.**DummyPartition**(*tri*)
Bases: object

Dummy class to generate tri.

partition(*as_array=True*)
Return the tri index.

class mlens.utils.dummy.**InitMixin**
Bases: object

Mixin to make a mlens ensemble behave as Scikit-learn estimator.

Scikit-learn expects an estimator to be fully initialized when instantiated, but an ML-Ensemble estimator requires layers to be initialized before calling fit or predict makes sense.

InitMixin is intended to be used to create temporary test classes of proper mlens ensemble classes that are identical to the parent class except that __init__ will also initialize one layer with one estimator, and if applicable one meta estimator.

The layer estimator and the meta estimator are both the dummy AverageRegressor class to minimize complexity and avoids raising errors due to the estimators in the layers.

To create a testing class, modify the __init__ of the test class to call super().__init__ as in the example below.

Examples

Assert the `SuperLearner` passes the Scikit-learn estimator test

```
>>> from sklearn.utils.estimator_checks import check_estimator
>>> from mlens.ensemble import SuperLearner
>>> from mlens.utils.dummy import InitMixin
>>>
>>> class TestSuperLearner(InitMixin, SuperLearner):
...
...     def __init__(self):
...         super(TestSuperLearner, self).__init__()
>>>
>>> check_estimator(TestSuperLearner)
```

class mlens.utils.dummy.**LayerGenerator**
Bases: object

Class for generating architectures of various types.

get_layer(*kls*, *proba*, *preprocessing*, **args*, ***kwargs*)
Generate a layer instance.

Parameters

- **kls** (*str*) – class type
- **proba** (*bool*) – whether to set `proba` to `True`
- **preprocessing** (*bool*) – layer with preprocessing cases

get_layer_container(*kls*, *proba*, *preprocessing*, **args*, ***kwargs*)
Generate a layer container instance.

Parameters

- **kls** (*str*) – class type
- **proba** (*bool*) – whether to set `proba` to `True`
- **preprocessing** (*bool*) – layer with preprocessing cases

static load_indexer(*kls*, *args*, *kwargs*)
Load indexer and return remaining kwargs

class mlens.utils.dummy.**LogisticRegression**(*offset=0*)
Bases: *mlens.utils.dummy.OLS*

No frill Logistic Regressor w. one-vs-rest estimation of P(label).

MWE of a Scikit-learn classifier.

LogisticRegression is a simple classifier estimator designed for transparency in unit testing. It implements a Logistic Regression with one-vs-rest strategy of classification.

The estimator is a wrapper around the *OLS*. The OLS prediction is squashed using the Sigmoid function, and classification is done by picking the label with the highest probability.

The `offset` option allows the user to offset weights in the OLS by a scalar value, if different instances should be differentiated in their predictions.

Examples

Asserting the LogisticRegression passes the Scikit-learn estimator test

```
>>> from sklearn.utils.estimator_checks import check_estimator
>>> from mlens.utils.dummy import LogisticRegression
>>> check_estimator(LogisticRegression)
```

Comparison with Scikit-learn's LogisticRegression

```
>>> from mlens.utils.dummy import LogisticRegression as mlensL
>>> from sklearn.linear_model import LogisticRegression as sklearnL
>>> from sklearn.datasets import make_classification
>>> X, y = make_classification()
>>>
>>> slr = sklearnL()
>>> slr.fit(X, y)
>>>
>>> mlr = mlensL()
>>> mlr.fit(X, y)
>>>
>>> (mlr.predict(X) == slr.predict(X)).sum() / y.shape
array([ 0.98])
```

fit (*X*, *y*)
Fit one model per label.

predict (*X*)
Get label predictions.

predict_proba (*X*)
Get probability predictions.

class mlens.utils.dummy.**OLS** (*offset=0*)
Bases: mlens.externals.sklearn.base.BaseEstimator

No frills vanilla OLS estimator implemented through the normal equation.

MWE of a Scikit-learn estimator.

OLS is a simple estimator designed to allow for total control over predictions in unit testing. It implements OLS through the Normal Equation, no learning takes place. The `offset` option allows the user to offset weights by a scalar value, if different instances should be differentiated in their predictions.

Parameters **offset** (*float (default = 0)*) – scalar value to add to the coefficient vector after fitting.

Examples

Asserting the OLS passes the Scikit-learn estimator test

```
>>> from sklearn.utils.estimator_checks import check_estimator
>>> from mlens.utils.dummy import OLS
>>> check_estimator(OLS)
```

OLS comparison with Scikit-learn's LinearRegression

```
>>> from numpy.testing import assert_array_equal
>>> from mlens.utils.dummy import OLS
>>> from sklearn.linear_model import LinearRegression
>>> from sklearn.datasets import load_boston
>>> X, y = load_boston(True)
>>>
>>> lr = LinearRegression(False)
>>> lr.fit(X, y)
>>>
>>> ols = OLS()
>>> ols.fit(X, y)
>>>
>>> assert_array_equal(lr.coef_, ols.coef_)
```

fit (*X*, *y*)
Fit coefficient vector.

predict (*X*)
Predict with fitted weights.

class mlens.utils.dummy.**Scale**(*copy=True*)
Bases: mlens.externals.sklearn.base.BaseEstimator, mlens.externals.sklearn.base.TransformerMixin

Removes the a learnt mean in a column-wise manner in an array.

MWE of a Scikit-learn transformer, to be used for unit-tests of ensemble classes.

Parameters **copy** (*bool (default = True)*) – Whether to copy X before transforming.

Examples

Asserting *Scale* passes the Scikit-learn estimator test

```
>>> from sklearn.utils.estimator_checks import check_estimator
>>> from mlens.utils.dummy import Scale
>>> check_estimator(Scale)
```

Scaling elements

```
>>> from numpy import arange
>>> from mlens.utils.dummy import Scale
>>> X = arange(6).reshape(3, 2)
>>> X[:, 1] *= 2
>>> print('X:')
>>> print('%r' % X)
>>> print('Scaled:')
>>> S = Scale().fit_transform(X)
>>> print('%r' % S)
X:
array([[ 0,  2],
       [ 2,  6],
       [ 4, 10]])
Scaled:
array([[-2., -4.],
       [ 0.,  0.],
       [ 2.,  4.]])
```

fit (*X*, *y=None*)
Estimate mean.

Parameters

- **X** (*array-like*) – training data to fit transformer on.
- **y** (*array-like or None*) – pass through for pipeline.

transform (*X*)
Transform array by adjusting all elements with scale.

Parameters **X** (*ndarray*) – matrix to transform.

mlens.utils.dummy.**layer_fit** (*layer*, *cache*, *F*, *wf*)
Test the layer's fit method.

mlens.utils.dummy.**layer_predict** (*layer*, *cache*, *P*, *wp*)
Test the layer's predict method.

mlens.utils.dummy.**layer_transform** (*layer*, *cache*, *F*)
Test the layer's transform method.

mlens.utils.dummy.**lc_feature_prop** (*lc*, *X*, *y*, *F*)
Test input feature propagation.

mlens.utils.dummy.**lc_fit** (*lc*, *X*, *y*, *F*, *wf*)
Test the layer containers fit method.

mlens.utils.dummy.**lc_from_csv** (*lc*, *cache*, *X*, *y*, *F*, *wf*, *P*, *wp*)
Fit a layer container from file path to csv.

mlens.utils.dummy.**lc_from_file** (*lc*, *cache*, *X*, *y*, *F*, *wf*, *P*, *wp*)
Fit a layer container from file path to numpy array.

mlens.utils.dummy.**lc_predict** (*lc*, *X*, *P*, *wp*)
Test the layer containers predict method.

mlens.utils.dummy.**lc_transform** (*lc*, *X*, *F*)
Test the layer containers transform method.

mlens.utils.exceptions module

ML-ENSEMBLE

Exception handling classes.

exception mlens.utils.exceptions.**DataConversionWarning**
Bases: UserWarning

Warning used to notify implicit data conversions happening in the code.

This warning occurs when some input data needs to be converted or interpreted in a way that may not match the user's expectations.

For example, this warning may occur when the user

- passes an integer array to a function which expects float input and will convert the input
- requests a non-copying operation, but a copy is required to meet the implementation's data-type expectations;
- passes an input whose shape can be interpreted ambiguously.

Changed in version 0.18: Moved from sklearn.utils.validation.

Note: imported from Scikit-learn for validation compatibility.

---

exception mlens.utils.exceptions.**EfficiencyWarning**
Bases: UserWarning

Warning used to notify the user of inefficient computation.

This warning notifies the user that the efficiency may not be optimal due to some reason which may be included as a part of the warning message. This may be subclassed into a more specific Warning class.

New in version 0.18.

Note: imported from Scikit-learn for validation compatibility

---

exception mlens.utils.exceptions.**FitFailedError**
Bases: RuntimeError, TypeError

Error for failed estimator 'fit' call.

Inherits type error to accommodate Scikit-learn expectation of a TypeError on failed array checks in estimators.

exception mlens.utils.exceptions.**FitFailedWarning**
Bases: RuntimeWarning

Warning for a failed estimator 'fit' call.

exception mlens.utils.exceptions.**InputDataWarning**
Bases: UserWarning

Warning used to notify that an array does not behave as expected.

Raised if data looks suspicious, but not outright fatal. Used sparingly, as it is often better to raise an error if input does not look like expected. Debugging corrupt data during parallel estimation is difficult and requires knowledge of backend operations.

exception mlens.utils.exceptions.**LayerSpecificationError**
Bases: TypeError, ValueError

Error class for incorrectly specified layers.

exception mlens.utils.exceptions.**LayerSpecificationWarning**
Bases: UserWarning

Warning class if layer has been specified in a dubious form.

This warning is raised when the input does not look like expected, but is not fatal and a best guess of how to fix it will be made.

exception mlens.utils.exceptions.**NonBLASDotWarning**
Bases: *mlens.utils.exceptions.EfficiencyWarning*

Warning used when the dot operation does not use BLAS.

FROM SCIKIT-LEARN

This warning is used to notify the user that BLAS was not used for dot operation and hence the efficiency may be affected.

Changed in version 0.18: Moved from sklearn.utils.validation, extends EfficiencyWarning.

Note: imported from Scikit-learn for validation compatibility

exception mlens.utils.exceptions.**NotFittedError**
Bases: ValueError, AttributeError

Error class for an ensemble or estimator that is not fitted yet

Raised when some method has been called that expects the instance to be fitted.

exception mlens.utils.exceptions.**ParallelProcessingError**
Bases: AttributeError, RuntimeError

Error class for fatal errors related to ParallelProcessing.

Can be subclassed for more specific error classes.

exception mlens.utils.exceptions.**ParallelProcessingWarning**
Bases: UserWarning

Warnings related to methods on ParallelProcessing.

Can be subclassed for more specific warning classes.

exception mlens.utils.exceptions.**PredictFailedError**
Bases: RuntimeError, TypeError

Error for a failed estimator 'predict' call.

Inherits type error to accommodate Scikit-learn expectation of a TypeError on failed array checks in estimators.

exception mlens.utils.exceptions.**PredictFailedWarning**
Bases: RuntimeWarning

Warning for a failed estimator 'predict' call.

mlens.utils.formatting module

ML-ENSEMBLE

author Sebastian Flennerhag

copyright 2017

licence MIT

Formatting of instance lists.

mlens.utils.formatting.**check_instances**(*instances*)
Helper to ensure all instances are named.

Check if instances is formatted as expected, and if not convert formatting or throw traceback error if impossible to anticipate formatting.

Parameters **instances** (*iterable*) – instance iterable to test.

Returns **formatted** – formatted instances object. Will be formatted as a dict if preprocessing cases are detected, otherwise as a list. The dict will contain lists identical to those in the single preprocessing case. Each list is of the form [('name', instance] and no names overlap.

Return type list or dict

Raises *LayerSpecificationError :* – Raises error if formatting fails, which is most likely due to wrong ordering of tuple entries, or wrong argument in the wrong position.

mlens.utils.utils module

ML-ENSEMBLE

author Sebastian Flennerhag

copyright 2017

licence MIT

class `mlens.utils.utils.`**`CMLog`**(*verbose=False*)

Bases: `object`

CPU and Memory logger.

Class for starting a monitor job of CPU and memory utilization in the background in a Python script. The `monitor` class records the `cpu_percent`, `rss` and `vms` as collected by the psutil library for the parent process' pid.

CPU usage and memory utilization are stored as attributes in numpy arrays.

Examples

```
>>> from time import sleep
>>> from mlens.utils.utils import CMLog
>>> cm = CMLog(verbose=True)
>>> cm.monitor(2, 0.5)
>>> _ = [i for i in range(10000000)]
>>>
>>> # Collecting before completion triggers a message but no error
>>> cm._collect()
>>>
>>> sleep(2)
>>> cm._collect()
>>> print('CPU usage:')
>>> cm.cpu
[CMLog] Monitoring for 2 seconds with checks every 0.5 seconds.
[CMLog] Job not finished. Cannot _collect yet.
[CMLog] Collecting... done. Read 4 lines in 0.000 seconds.
CPU usage:
array([ 50. ,  22.4,   6. ,  11.9])
```

Raises *ImportError :* – Depends on psutil. If not installed, raises ImportError on instantiation.

Parameters **`verbose`** (*`bool`*) – whether to notify of job start.

`collect`()

Collect monitored data.

Once a monitor job finishes, call `_collect` to read the CPU and memory usage into python objects in the current process. If called before the job finishes, _collect issues a print statement to try again later, but no warning or error is raised.

`monitor`(*stop=None*, *ival=0.1*, *kill=True*)

Start monitoring CPU and memory usage.

Parameters

- **stop** (*float or None (default = None)*) – seconds to monitor for. If None, monitors until `_collect` is called.
- **ival** (*float (default=0.1)*) – interval of monitoring.
- **kill** (*bool (default = True)*) – whether to kill the monitoring job if `_collect` is called before timeout (`stop`). If set to False, calling `_collect` will cause the instance to wait until the job completes.

mlens.utils.utils.**kwarg_parser**(*func*, *kwargs*)
Utility function for parsing keyword arguments

mlens.utils.utils.**pickle_load**(*name*)
Utility function for loading pickled object

mlens.utils.utils.**pickle_save**(*obj*, *name*)
Utility function for pickling an object

mlens.utils.utils.**print_time**(*t0*, *message=''*, ***kwargs*)
Utility function for printing time

mlens.utils.utils.**safe_print**(**objects*, ***kwargs*)
Safe print function for backwards compatibility.

mlens.utils.validation module

ML-ENSEMBLE

author Sebastian Flennerhag

copyright 2017

license MIT

Input validation module. Builds on Scikit-learns `validation` module, but extends it to a soft check that issues warnings but don't force change the inputs.

mlens.utils.validation.**check_all_finite**(*X*)
Return False if X contains NaN or infinity.

mlens.utils.validation.**check_inputs**(*X*, *y=None*, *check_level=0*)
Pre-checks on input arrays X and y.

Checks input data according to `check_level` to ensure format is roughly in line with what a typical estimator expects.

If `check_level = 0` this test is turned off.

Parameters

- **X** (*nd-array, list or sparse matrix*) – Input data.
- **y** (*nd-array, list or sparse matrix*) – Labels.
- **check_level** (*int (default = 2)*) – level of strictness in checking input arrays.
  - `check_level = 0` no checks, returns X, y
  - `check_level` = 1 will raises warnings if any non-critical test fails. Returns boolean FAIL flag.

– `check_level = 2` will impose Scikit-learn array check, which converts `X` and `y` to numpy arrays and raises error if conversion fails.

Returns

- **FAIL** (*fail flag, optional*) – boolean for whether any test failed. Returned if `check_level = 1`
- **X_converted** (*numpy array, optional*) – The converted and validated X. Returned if `check_level = 2`
- **y_converted** (*numpy array, optional*) – The converted and validated y. Returned if `check_level = 2`.
- **random_state** (*object, optional*) – numpy RandomState object.

`mlens.utils.validation.`**`soft_check_1d`**(*y*, *y_numeric*, *estimator*)

Check if y is numeric, finite and one-dimensional.

`mlens.utils.validation.`**`soft_check_array`**(*array*, *accept_sparse=True*, *dtype=None*, *ensure_2d=True*, *force_all_finite=True*, *allow_nd=True*, *ensure_min_samples=1*, *ensure_min_features=1*, *estimator=None*)

Input validation on an array, list, sparse matrix or similar.

Like Scikit-learn's `check_array` , but issues warnings on failed tests and do no forced array conversion.

Parameters

- **`array`** (*`array-like`*) – Input object, expected to be array-like, to check / convert.
- **`accept_sparse`** (*`string, list of string or None (default=None)`*) – String[s] representing allowed sparse matrix formats, such as 'csc', 'csr', etc. None means that sparse matrix input will raise an error. If the input is sparse but not in the allowed format, it will be converted to the first listed format.
- **`dtype`** (*`string, type, list of types or None (default="numeric")`*) – Data type of result. If None, the dtype of the input is preserved. If "numeric", warning is raised if array.dtype is object. If dtype is a list of types, warning is raised if array.dtype is not a member of the list.
- **`force_all_finite`** (*`boolean (default=True)`*) – Whether to raise an error on np.inf and np.nan in X.
- **`ensure_2d`** (*`boolean (default=True)`*) – Whether to warn if X is not at least 2d.
- **`allow_nd`** (*`boolean (default=False)`*) – Whether to allow X.ndim > 2.
- **`ensure_min_samples`** (*`int (default=1)`*) – Make sure that the array has a minimum number of samples in its first axis (rows for a 2D array). Setting to 0 disables this check.
- **`ensure_min_features`** (*`int (default=1)`*) – Make sure that the 2D array has some minimum number of features (columns). The default value of 1 rejects empty datasets. This check is only enforced when the input data has effectively 2 dimensions or is originally 1D and `ensure_2d` is True. Setting to 0 disables this check.
- **`estimator`** (*`str or estimator instance (default=None)`*) – If passed, include the name of the estimator in warning messages.

Returns **CHANGE** – Whether X should be changed.

Return type bool

mlens.utils.validation.**soft_check_x_y**(*X*, *y*, *accept_sparse=True*, *dtype=None*, *force_all_finite=True*, *ensure_2d=True*, *allow_nd=True*, *multi_output=False*, *ensure_min_samples=1*, *ensure_min_features=1*, *y_numeric=False*, *estimator=None*)

Input validation before estimation.

Checks X and y for consistent length, and X 2d and y 1d. Standard input checks are only applied to y, such as checking that y does not have np.nan or np.inf targets. For multi-label y, set multi_output=True to allow 2d and sparse y. Raises warnings if the dtype is object.

Parameters

- **X** (*nd-array, list or sparse matrix*) – Input data.
- **y** (*nd-array, list or sparse matrix*) – Labels.
- **accept_sparse** (*string, list of string or None (default=None)*) – String[s] representing allowed sparse matrix formats, such as 'csc', 'csr', etc. None means that sparse matrix input will raise an error. If the input is sparse but not in the allowed format, it will be converted to the first listed format.
- **dtype** (*string, type, list of types or None (default="numeric")*) – Data type of result. If None, the dtype of the input is preserved. If "numeric", dtype is preserved unless array.dtype is object. If dtype is a list of types, conversion on the first type is only performed if the dtype of the input is not in the list.
- **force_all_finite** (*boolean (default=True)*) – Whether to raise an error on np.inf and np.nan in X. This parameter does not influence whether y can have np.inf or np.nan values.
- **ensure_2d** (*boolean (default=True)*) – Whether to make X at least 2d.
- **allow_nd** (*boolean (default=False)*) – Whether to allow X.ndim > 2.
- **multi_output** (*boolean (default=False)*) – Whether to allow 2-d y (array or sparse matrix). If false, y will be validated as a vector. y cannot have np.nan or np.inf values if multi_output=True.
- **ensure_min_samples** (*int (default=1)*) – Make sure that X has a minimum number of samples in its first axis (rows for a 2D array).
- **ensure_min_features** (*int (default=1)*) – Make sure that the 2D array has some minimum number of features (columns). The default value of 1 rejects empty datasets. This check is only enforced when X has effectively 2 dimensions or is originally 1D and `ensure_2d` is True. Setting to 0 disables this check.
- **y_numeric** (*boolean (default=False)*) – Whether to ensure that y has a numeric type. If dtype of y is object, it is converted to float64. Should only be used for regression algorithms.
- **estimator** (*str or estimator instance (default=None)*) – If passed, include the name of the estimator in warning messages.

Returns

- **X_converted** (*object*) – The converted and validated X.
- **y_converted** (*object*) – The converted and validated y.

Module contents

ML-ENSEMBLE

author Sebastian Flennerhag

copyright 2017

licence MIT

mlens.utils.**check_inputs**(*X*, *y=None*, *check_level=0*)

Pre-checks on input arrays X and y.

Checks input data according to `check_level` to ensure format is roughly in line with what a typical estimator expects.

If `check_level = 0` this test is turned off.

Parameters

- **X** (*nd-array, list or sparse matrix*) – Input data.
- **y** (*nd-array, list or sparse matrix*) – Labels.
- **check_level** (*int (default = 2)*) – level of strictness in checking input arrays.
  - `check_level = 0` no checks, returns X, y
  - `check_level` = 1 will raises warnings if any non-critical test fails. Returns boolean FAIL flag.
  - `check_level = 2` will impose Scikit-learn array check, which converts `X` and `y` to numpy arrays and raises error if conversion fails.

Returns

- **FAIL** (*fail flag, optional*) – boolean for whether any test failed. Returned if `check_level = 1`
- **X_converted** (*numpy array, optional*) – The converted and validated X. Returned if `check_level = 2`
- **y_converted** (*numpy array, optional*) – The converted and validated y. Returned if `check_level = 2`.
- **random_state** (*object, optional*) – numpy RandomState object.

mlens.utils.**check_instances**(*instances*)

Helper to ensure all instances are named.

Check if `instances` is formatted as expected, and if not convert formatting or throw traceback error if impossible to anticipate formatting.

Parameters **instances** (*iterable*) – instance iterable to test.

Returns **formatted** – formatted `instances` object. Will be formatted as a dict if preprocessing cases are detected, otherwise as a list. The dict will contain lists identical to those in the single preprocessing case. Each list is of the form `[('name', instance]` and no names overlap.

Return type list or dict

Raises *LayerSpecificationError :* – Raises error if formatting fails, which is most likely due to wrong ordering of tuple entries, or wrong argument in the wrong position.

mlens.utils.**check_is_fitted**(*estimator*, *attr*)

Check that ensemble has been fitted.

Parameters

- **estimator** (*estimator instance*) – ensemble instance to check.
- **attr** (*str*) – attribute to assert existence of.

mlens.utils.**check_ensemble_build**(*inst*, *attr='layers'*)
Check that layers have been instantiated.

mlens.utils.**assert_correct_format**(*estimators*, *preprocessing*)
Initial check to assert layer can be constructed.

mlens.utils.**check_initialized**(*inst*)
Check if a ParallelProcessing instance is initialized properly.

mlens.utils.**pickle_save**(*obj*, *name*)
Utility function for pickling an object

mlens.utils.**pickle_load**(*name*)
Utility function for loading pickled object

mlens.utils.**print_time**(*t0*, *message=''*, ***kwargs*)
Utility function for printing time

mlens.utils.**safe_print**(**objects*, ***kwargs*)
Safe print function for backwards compatibility.

class mlens.utils.**CMLog**(*verbose=False*)
Bases: object

CPU and Memory logger.

Class for starting a monitor job of CPU and memory utilization in the background in a Python script. The monitor class records the cpu_percent, rss and vms as collected by the psutil library for the parent process' pid.

CPU usage and memory utilization are stored as attributes in numpy arrays.

Examples

```
>>> from time import sleep
>>> from mlens.utils.utils import CMLog
>>> cm = CMLog(verbose=True)
>>> cm.monitor(2, 0.5)
>>> _ = [i for i in range(10000000)]
>>>
>>> # Collecting before completion triggers a message but no error
>>> cm._collect()
>>>
>>> sleep(2)
>>> cm._collect()
>>> print('CPU usage:')
>>> cm.cpu
[CMLog] Monitoring for 2 seconds with checks every 0.5 seconds.
[CMLog] Job not finished. Cannot _collect yet.
[CMLog] Collecting... done. Read 4 lines in 0.000 seconds.
CPU usage:
array([ 50. ,  22.4,   6. ,  11.9])
```

Raises *ImportError :* – Depends on psutil. If not installed, raises ImportError on instantiation.

Parameters **verbose** (*bool*) – whether to notify of job start.

collect()
Collect monitored data.

Once a monitor job finishes, call _collect to read the CPU and memory usage into python objects in the current process. If called before the job finishes, _collect issues a print statement to try again later, but no warning or error is raised.

monitor(*stop=None*, *ival=0.1*, *kill=True*)
Start monitoring CPU and memory usage.

Parameters

- **stop** (*float or None (default = None)*) – seconds to monitor for. If None, monitors until _collect is called.
- **ival** (*float (default=0.1)*) – interval of monitoring.
- **kill** (*bool (default = True)*) – whether to kill the monitoring job if _collect is called before timeout (stop). If set to False, calling _collect will cause the instance to wait until the job completes.

mlens.utils.**kwarg_parser**(*func*, *kwargs*)
Utility function for parsing keyword arguments

mlens.visualization package

Submodules

mlens.visualization.correlations module

ML-ENSEMBLE

author Sebastian Flennerhag

copyright 2017

licence MIT

Correlation plots.

mlens.visualization.correlations.**clustered_corrmap**(*corr*, *cls*, *label_attr_name='labels_'*, *figsize=(10, 8)*, *annotate=False*, *inflate=False*, *linewidths=0.5*, *cbar_kws='default'*, *show=True*, *title_fontsize=14*, *title_name='Clustered correlation heatmap'*, *ax=None*, ***kwargs*)
Function for plotting a clustered correlation heatmap.

Parameters

- **corr** (*array-like of shape = [n_features, n_features]*) – Input correlation matrix. Pass a pandas DataFrame for axis labels.
- **cls** (*instance*) – cluster estimator with a fit method and cluster labels stored as an attribute as specified by the label_attr_name parameter.

- **label_attr_name** (*str*) – name of attribute that contains cluster labels.
- **figsize** (*tuple (default = (10, 8))*) – Size of figure.
- **annotate** (*bool (default = True)*) – Whether to print the correlation coefficients.
- **inflate** (*bool (default = True)*) – Whether to inflate correlation coefficients to a 0-100 scale. Avoids decimal points in the figure, which often appears very cluttered otherwise.
- **linewidths** (*float (default = .5)*) – with of line separating each coordinate square.
- **cbar_kws** (*dict, str (default = 'default')*) – Optional arguments to color bar.
- **title_name** (*str*) – Figure title.
- **title_fontsize** (*int*) – size of title.
- **show** (*bool (default = True)*) – whether to print figure using matplotlib.pyplot.show.
- **ax** (*object, optional*) – axis to attach plot to.
- ****kwargs** (*optional*) – Other optional arguments to sns heatmap.

See also:

mlens.visualization.corrmat

mlens.visualization.correlations.**corr_X_y**(*X*, *y*, *top=5*, *figsize=(10, 8)*, *fontsize=12*, *hspace=None*, *no_ticks=True*, *label_rotation=0*, *show=True*)

Function for plotting input feature correlations with output.

Output figure shows all correlations as well as top pos and neg.

Parameters

- **X** (*pandas DataFrame of shape = [n_samples, n_features]*) – Input data.
- **y** (*pandas Series of shape = [n_samples,]*) – training labels.
- **top** (*int*) – number of features to show in top pos and neg graphs.
- **figsize** (*tuple (default = (10, 8))*) – Size of figure.
- **hspace** (*float, optional*) – whitespace between top row of figures and bottom figure.
- **fontsize** (*int*) – font size of subplot titles.
- **no_ticks** (*bool (default = False)*) – whether to remove ticklabels from full correlation plot.
- **label_rotation** (*float (default = 0)*) – rotation of labels
- **show** (*bool (default = True)*) – whether to print figure using matplotlib.pyplot.show.

Returns **ax** – axis object.

Return type object

mlens.visualization.correlations.**corrmat**(*corr*, *figsize=(11, 9)*, *annotate=True*, *inflate=True*, *linewidths=0.5*, *cbar_kws='default'*, *show=True*, *ax=None*, *title='Correlation Matrix'*, *title_font_size=14*, ***kwargs*)

Function for generating color-coded correlation triangle.

Parameters

- **corr** (*array-like of shape = [n_features, n_features]*) – Input correlation matrix. Pass a pandas `DataFrame` for axis labels.
- **figsize** (*tuple (default = (11, 9))*) – Size of printed figure.
- **annotate** (*bool (default = True)*) – Whether to print the correlation coefficients.
- **inflate** (*bool (default = True)*) – Whether to inflate correlation coefficients to a 0-100 scale. Avoids decimal points in the figure, which often appears very cluttered otherwise.
- **linewidths** (*float*) – with of line separating each coordinate square.
- **cbar_kws** (*dict, str (default = 'default')*) – Optional arguments to color bar. The default options, 'default', passes the `shrink` parameter to fit colorbar standard figure frame.
- **show** (*bool (default = True)*) – whether to print figure using `matplotlib.pyplot.show`.
- **title** (*str*) – figure title if shown.
- **title_font_size** (*int*) – title font size.
- **ax** (*object, optional*) – axis to attach plot to.
- ****kwargs** (*optional*) – Other optional arguments to sns heatmap.

Returns **ax** – axis object.

Return type object

See also:

mlens.visualization.clustered_corrmap

mlens.visualization.var_analysis module

ML-ENSEMBLE

author Sebastian Flennerhag

copyright 2017

licence MIT

Explained variance plots.

mlens.visualization.var_analysis.**exp_var_plot**(*X*, *estimator*, *figsize=(10, 8)*, *buffer=0.01*, *set_labels=True*, *title='Explained variance ratio'*, *title_font_size=14*, *show=True*, *ax=None*, ***kwargs*)

Function to plot the explained variance using PCA.

Parameters

- **X** (*array-like of shape = [n_samples, n_features]*) – input matrix to be used for prediction.
- **estimator** (*class*) – PCA estimator, not initiated, assumes a Scikit-learn API.
- **figsize** (*tuple (default = (10, 8))*) – Size of figure.
- **buffer** (*float (default = 0.01)*) – For creating a buffer around the edges of the graph. The buffer added is calculated as `num_components * buffer`, where `num_components` determine the length of the x-axis.
- **set_labels** (*bool*) – whether to set axis labels.
- **title** (*str*) – figure title if shown.
- **title_font_size** (*int*) – title font size.
- **show** (*bool (default = True)*) – whether to print figure using `matplotlib.pyplot.show`.
- **ax** (*object, optional*) – axis to attach plot to.
- ****kwargs** (*optional*) – optional arguments passed to the `matplotlib.pyplot.step` function.

Returns **ax** – if `ax` was specified, returns `ax` with plot attached.

Return type optional

mlens.visualization.var_analysis.**pca_comp_plot**(*X*, *y=None*, *figsize=(10, 8)*, *title='Principal Components Comparison'*, *title_font_size=14*, *show=True*, ***kwargs*)

Function for comparing PCA analysis.

Function compares across 2 and 3 dimensions and linear and rbf kernels.

Parameters

- **X** (*array-like of shape = [n_samples, n_features]*) – input matrix to be used for prediction.
- **y** (*array-like of shape = [n_samples,] or None (default = None)*) – training labels to be used for color highlighting.
- **figsize** (*tuple (default = (10, 8))*) – Size of figure.
- **title** (*str*) – figure title if shown.
- **title_font_size** (*int*) – title font size.
- **show** (*bool (default = True)*) – whether to print figure `matplotlib.pyplot.show`.
- ****kwargs** (*optional*) – optional arguments to pass to *mlens.visualization.pca_plot*.

Returns axis object.

Return type ax

See also:

mlens.visualization.pca_plot

mlens.visualization.var_analysis.**pca_plot**(*X*, *estimator*, *y=None*, *cmap=None*, *figsize=(10, 8)*, *title='Principal Components Analysis'*, *title_font_size=14*, *show=True*, *ax=None*, ***kwargs*)

Function to plot a PCA analysis of 1, 2, or 3 dims.

Parameters

- **X** (*array-like of shape = [n_samples, n_features]*) – matrix to perform PCA analysis on.
- **estimator** (*instance*) – PCA estimator. Assumes a Scikit-learn API.
- **y** (*array-like of shape = [n_samples,] or None (default = None)*) – training labels to be used for color highlighting.
- **cmap** (*object, optional*) – cmap object to pass to matplotlib.pyplot.scatter.
- **figsize** (*tuple (default = (10, 8))*) – Size of figure.
- **title** (*str*) – figure title if shown.
- **title_font_size** (*int*) – title font size.
- **show** (*bool (default = True)*) – whether to print figure matplotlib.pyplot.show.
- **ax** (*object, optional*) – axis to attach plot to.
- ****kwargs** (*optional*) – arguments to pass to matplotlib.pyplot.scatter.

Returns **ax** – if ax was specified, returns ax with plot attached.

Return type optional

Module contents

ML-ENSEMBLE

author Sebastian Flennerhag

copyright 2017

license MIT

mlens.visualization.**corrmat**(*corr*, *figsize=(11, 9)*, *annotate=True*, *inflate=True*, *linewidths=0.5*, *cbar_kws='default'*, *show=True*, *ax=None*, *title='Correlation Matrix'*, *title_font_size=14*, ***kwargs*)

Function for generating color-coded correlation triangle.

Parameters

- **corr** (*array-like of shape = [n_features, n_features]*) – Input correlation matrix. Pass a pandas DataFrame for axis labels.
- **figsize** (*tuple (default = (11, 9))*) – Size of printed figure.
- **annotate** (*bool (default = True)*) – Whether to print the correlation coefficients.
- **inflate** (*bool (default = True)*) – Whether to inflate correlation coefficients to a 0-100 scale. Avoids decimal points in the figure, which often appears very cluttered otherwise.

- **linewidths** (*float*) – with of line separating each coordinate square.
- **cbar_kws** (*dict, str (default = 'default')*) – Optional arguments to color bar. The default options, 'default', passes the `shrink` parameter to fit colorbar standard figure frame.
- **show** (*bool (default = True)*) – whether to print figure using `matplotlib.pyplot.show`.
- **title** (*str*) – figure title if shown.
- **title_font_size** (*int*) – title font size.
- **ax** (*object, optional*) – axis to attach plot to.
- ****kwargs** (*optional*) – Other optional arguments to sns heatmap.

Returns **ax** – axis object.

Return type object

See also:

mlens.visualization.clustered_corrmap

mlens.visualization.**clustered_corrmap**(*corr*, *cls*, *label_attr_name='labels_'*, *figsize=(10, 8)*, *annotate=False*, *inflate=False*, *linewidths=0.5*, *cbar_kws='default'*, *show=True*, *title_fontsize=14*, *title_name='Clustered correlation heatmap'*, *ax=None*, ***kwargs*)

Function for plotting a clustered correlation heatmap.

Parameters

- **corr** (*array-like of shape = [n_features, n_features]*) – Input correlation matrix. Pass a pandas `DataFrame` for axis labels.
- **cls** (*instance*) – cluster estimator with a `fit` method and cluster labels stored as an attribute as specified by the `label_attr_name` parameter.
- **label_attr_name** (*str*) – name of attribute that contains cluster labels.
- **figsize** (*tuple (default = (10, 8))*) – Size of figure.
- **annotate** (*bool (default = True)*) – Whether to print the correlation coefficients.
- **inflate** (*bool (default = True)*) – Whether to inflate correlation coefficients to a 0-100 scale. Avoids decimal points in the figure, which often appears very cluttered otherwise.
- **linewidths** (*float (default = .5)*) – with of line separating each coordinate square.
- **cbar_kws** (*dict, str (default = 'default')*) – Optional arguments to color bar.
- **title_name** (*str*) – Figure title.
- **title_fontsize** (*int*) – size of title.
- **show** (*bool (default = True)*) – whether to print figure using `matplotlib.pyplot.show`.
- **ax** (*object, optional*) – axis to attach plot to.
- ****kwargs** (*optional*) – Other optional arguments to sns heatmap.

See also:

mlens.visualization.corrmat

mlens.visualization.**corr_X_y**(*X*, *y*, *top=5*, *figsize=(10, 8)*, *fontsize=12*, *hspace=None*, *no_ticks=True*, *label_rotation=0*, *show=True*)

Function for plotting input feature correlations with output.

Output figure shows all correlations as well as top pos and neg.

Parameters

- **X** (*pandas DataFrame of shape = [n_samples, n_features]*) – Input data.
- **y** (*pandas Series of shape = [n_samples,]*) – training labels.
- **top** (*int*) – number of features to show in top pos and neg graphs.
- **figsize** (*tuple (default = (10, 8))*) – Size of figure.
- **hspace** (*float, optional*) – whitespace between top row of figures and bottom figure.
- **fontsize** (*int*) – font size of subplot titles.
- **no_ticks** (*bool (default = False)*) – whether to remove ticklabels from full correlation plot.
- **label_rotation** (*float (default = 0)*) – rotation of labels
- **show** (*bool (default = True)*) – whether to print figure using matplotlib.pyplot.show.

Returns **ax** – axis object.

Return type object

mlens.visualization.**pca_comp_plot**(*X*, *y=None*, *figsize=(10, 8)*, *title='Principal Components Comparison'*, *title_font_size=14*, *show=True*, ***kwargs*)

Function for comparing PCA analysis.

Function compares across 2 and 3 dimensions and linear and rbf kernels.

Parameters

- **X** (*array-like of shape = [n_samples, n_features]*) – input matrix to be used for prediction.
- **y** (*array-like of shape = [n_samples,] or None (default = None)*) – training labels to be used for color highlighting.
- **figsize** (*tuple (default = (10, 8))*) – Size of figure.
- **title** (*str*) – figure title if shown.
- **title_font_size** (*int*) – title font size.
- **show** (*bool (default = True)*) – whether to print figure matplotlib.pyplot.show.
- ****kwargs** (*optional*) – optional arguments to pass to *mlens.visualization.pca_plot*.

Returns axis object.

Return type ax

See also:

mlens.visualization.pca_plot

mlens.visualization.**pca_plot**(*X*, *estimator*, *y=None*, *cmap=None*, *figsize=(10, 8)*, *title='Principal Components Analysis'*, *title_font_size=14*, *show=True*, *ax=None*, ***kwargs*)

Function to plot a PCA analysis of 1, 2, or 3 dims.

Parameters

- **X** (*array-like of shape = [n_samples, n_features]*) – matrix to perform PCA analysis on.
- **estimator** (*instance*) – PCA estimator. Assumes a Scikit-learn API.
- **y** (*array-like of shape = [n_samples,] or None (default = None)*) – training labels to be used for color highlighting.
- **cmap** (*object, optional*) – cmap object to pass to matplotlib.pyplot.scatter.
- **figsize** (*tuple (default = (10, 8))*) – Size of figure.
- **title** (*str*) – figure title if shown.
- **title_font_size** (*int*) – title font size.
- **show** (*bool (default = True)*) – whether to print figure matplotlib.pyplot.show.
- **ax** (*object, optional*) – axis to attach plot to.
- ****kwargs** (*optional*) – arguments to pass to matplotlib.pyplot.scatter.

Returns **ax** – if ax was specified, returns ax with plot attached.

Return type optional

mlens.visualization.**exp_var_plot**(*X*, *estimator*, *figsize=(10, 8)*, *buffer=0.01*, *set_labels=True*, *title='Explained variance ratio'*, *title_font_size=14*, *show=True*, *ax=None*, ***kwargs*)

Function to plot the explained variance using PCA.

Parameters

- **X** (*array-like of shape = [n_samples, n_features]*) – input matrix to be used for prediction.
- **estimator** (*class*) – PCA estimator, not initiated, assumes a Scikit-learn API.
- **figsize** (*tuple (default = (10, 8))*) – Size of figure.
- **buffer** (*float (default = 0.01)*) – For creating a buffer around the edges of the graph. The buffer added is calculated as num_components * buffer, where num_components determine the length of the x-axis.
- **set_labels** (*bool*) – whether to set axis labels.
- **title** (*str*) – figure title if shown.
- **title_font_size** (*int*) – title font size.
- **show** (*bool (default = True)*) – whether to print figure using matplotlib.pyplot.show.
- **ax** (*object, optional*) – axis to attach plot to.

- ****kwargs** (*optional*) – optional arguments passed to the matplotlib.pyplot.step function.

Returns **ax** – if ax was specified, returns ax with plot attached.

Return type optional

Module contents

ML-ENSEMBLE

author Sebastian Flennerhag

copyright 2017

licence MIT

ML-Ensemble, a Python library for memory efficient parallelized ensemble learning.

ML Ensemble is licenced under *MIT* and is hosted on Github.

Python Module Index

m

Index

D

E

F

G

I

R

S

T

U

W

Y