stream-learn

Release 0.8.24

P. Ksieniewicz, P. Zyblewski

Apr 28, 2024

GETTING STARTED

1 Quick start guide **1**
- 1.1 Installation . . . 1
- 1.2 Preparing experiments . . . 1
- 1.3 Processing and understanding results . . . 2

2 Data Streams **5**
- 2.1 Stationary stream . . . 5
- 2.2 Streams containing concept drifts . . . 6
- 2.3 Class imbalance . . . 7
- 2.4 Mixing drift properties . . . 8

3 Stream Evaluators **9**
- 3.1 Test-Then-Train Evaluator . . . 9
- 3.2 Prequential Evaluator . . . 10
- 3.3 Metrics . . . 11

4 Classifier Ensembles **15**
- 4.1 Chunk-based Ensembles for Data Streams . . . 15
- 4.2 Online Ensembles for Data Streams . . . 17

5 Classifiers **19**
- 5.1 Accumulated Samples Classifier . . . 19
- 5.2 Sample-Weighted Meta Estimator . . . 19

6 Streams module **21**

7 Evaluators module **27**

8 Ensembles module **31**
- 8.1 Parameters . . . 32
- 8.2 Returns . . . 32
- 8.3 Parameters . . . 33
- 8.4 Returns . . . 33
- 8.5 Parameters . . . 34
- 8.6 Returns . . . 34
- 8.7 Parameters . . . 34
- 8.8 Returns . . . 35
- 8.9 Parameters . . . 35
- 8.10 Returns . . . 36
- 8.11 Parameters . . . 36
- 8.12 Returns . . . 36

8.13 Parameters . . . 37
8.14 Returns . . . 37
8.15 Parameters . . . 38
8.16 Returns . . . 38
8.17 Parameters . . . 39
8.18 Returns . . . 39
8.19 Parameters . . . 39
8.20 Returns . . . 40
8.21 Parameters . . . 40
8.22 Returns . . . 41
8.23 Parameters . . . 41
8.24 Returns . . . 41
8.25 Parameters . . . 42
8.26 Returns . . . 42
8.27 Parameters . . . 43
8.28 Returns . . . 43
8.29 Parameters . . . 44
8.30 Returns . . . 44
8.31 Parameters . . . 44
8.32 Returns . . . 45
8.33 Parameters . . . 46
8.34 Returns . . . 46
8.35 Parameters . . . 47
8.36 Returns . . . 47
8.37 Parameters . . . 48
8.38 Returns . . . 48
8.39 Parameters . . . 48
8.40 Returns . . . 49
8.41 Parameters . . . 49
8.42 Returns . . . 50
8.43 Parameters . . . 50
8.44 Returns . . . 50
8.45 Parameters . . . 51
8.46 Returns . . . 51
8.47 Parameters . . . 52
8.48 Returns . . . 52
8.49 Parameters . . . 53
8.50 Returns . . . 54
8.51 Parameters . . . 54
8.52 Returns . . . 54
8.53 Parameters . . . 56
8.54 Returns . . . 56
8.55 Parameters . . . 57
8.56 Returns . . . 57
8.57 Parameters . . . 58
8.58 Returns . . . 58
8.59 Parameters . . . 58
8.60 Returns . . . 59
8.61 Parameters . . . 60
8.62 Returns . . . 61
8.63 Parameters . . . 61
8.64 Returns . . . 61

9 Classifiers module **63**

9.1 Parameters . . . 64
9.2 Returns . . . 64
9.3 Parameters . . . 65
9.4 Returns . . . 65
9.5 Parameters . . . 65
9.6 Returns . . . 66
9.7 Parameters . . . 66
9.8 Returns . . . 66
9.9 Parameters . . . 67
9.10 Returns . . . 67
9.11 Parameters . . . 68
9.12 Returns . . . 68

10 Utils module **69**

11 Metrics module **71**

12 About us **75**

13 Citation policy **77**

14 Getting started **79**

15 API Documentation **81**

16 Examples **83**

Python Module Index **85**

Index **87**

CHAPTER

ONE

QUICK START GUIDE

1.1 Installation

To use the *stream-learn* package, it will be absolutely useful to install it. Fortunately, it is available in the PyPI repository, so you may install it using *pip*:

```
pip install -U stream-learn
```

You can also install the module cloned from Github using the setup.py file if you have a strange, but perhaps legitimate need:

```
git clone https://github.com/w4k2/stream-learn.git
cd stream-learn
make install
```

1.2 Preparing experiments

In order to conduct experiments, a declaration of four elements is necessary. The first is the estimator, which must be compatible with the `scikit-learn` API and, in addition, implement the `partial_fit()` method, allowing you to re-fit the already built model. For example, we'll use the standard *Gaussian Naive Bayes* algorithm:

```
from sklearn.naive_bayes import GaussianNB
clf = GaussianNB()
```

The next element is the data stream that we aim to process. In the example we will use a synthetic stream consisting of shocking number of 30 chunks and containing precisely one concept drift. We will prepare it using the `StreamGenerator` class of the `stream-learn` module:

```
from strlearn.streams import StreamGenerator
stream = StreamGenerator(n_chunks=30, n_drifts=1)
```

The third requirement of the experiment is to specify the metrics used in the evaluation of the methods. In the example, we will use the *accuracy* metric available in `scikit-learn` and the *balanced accuracy* from the `stream-learn` module:

```
from sklearn.metrics import accuracy_score
from strlearn.metrics import balanced_accuracy_score
metrics = [accuracy_score, balanced_accuracy_score]
```

The last necessary element of processing is the evaluator, i.e. the method of conducting the experiment. For example, we will choose the *Test-Then-Train* paradigm, described in more detail in User Guide. It is important to note, that we need to provide the metrics that we will use in processing at the point of initializing the evaluator. In the case of none metrics given, it will use default pair of *accuracy* and *balanced accuracy* scores:

```
from strlearn.evaluators import TestThenTrain
evaluator = TestThenTrain(metrics)
```

1.3 Processing and understanding results

Once all processing requirements have been met, we can proceed with the evaluation. To start processing, call the evaluator's process method, feeding it with the stream and classifier:

```
evaluator.process(stream, clf)
```

The results obtained are stored in the `scores` atribute of evaluator. If we print it on the screen, we may be able to observe that it is a three-dimensional numpy array with dimensions `(1, 29, 2)`.

- The first dimension is the **index of a classifier** submitted for processing. In the example above, we used only one model, but it is also possible to pass a tuple or list of classifiers that will be processed in parallel (See User guide:evaluators).
- The second dimension specifies the **instance of evaluation**, which in the case of *Test-Then-Train* methodology directly means the index of the processed chunk.
- The third dimension indicates the **metric** used in the processing.

Using this knowledge, we may finally try to illustrate the results of our simple experiment in the form of a plot:

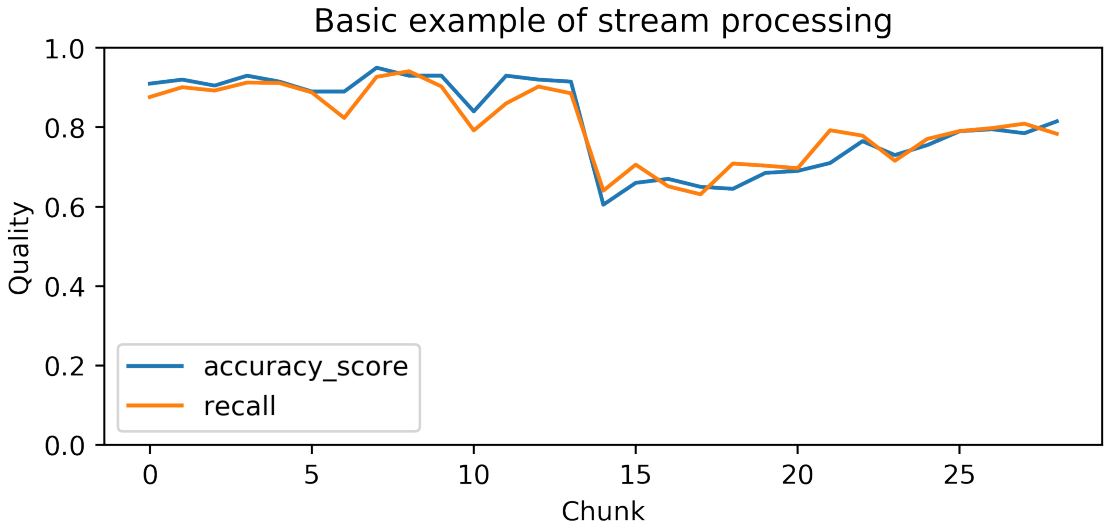
```
import matplotlib.pyplot as plt

plt.figure(figsize=(6,3))

for m, metric in enumerate(metrics):
    plt.plot(evaluator.scores[0, :, m], label=metric.__name__)

plt.title("Basic example of stream processing")
plt.ylim(0, 1)
plt.ylabel('Quality')
plt.xlabel('Chunk')

plt.legend()
```


Basic example of stream processing
Quality
Chunk
accuracy_score
recall
1.0
0.8
0.6
0.4
0.2
0.0
0
5
10
15
20
25

CHAPTER

TWO

DATA STREAMS

A key element of the `stream-learn` package is a generator that allows to prepare a replicable (according to the given `random_state` value) classification dataset with class distribution changing over the course of stream, with base concepts build on a default class distributions for the `scikit-learn` package from the `make_classification()` function. These types of distributions try to reproduce the rules for generating the `Madelon` set. The `StreamGenerator` is capable of preparing any variation of the data stream known in the general taxonomy of data streams.

2.1 Stationary stream

The simplest variation of data streams are *stationary streams*. They contain one basic concept, static for the whole course of the processing. Chunks differ from each other in terms of the patterns inside, but the decision boundaries of the models built on them should not be statistically different. This type of stream may be generated with a clean generator call, without any additional parameters.

```
StreamGenerator()
```

The above illustration contains the series of scatter plots for a two-dimensional stationary stream with the binary problem. The `StreamGenerator` class in the initializer accepts almost all standard attributes of the `make_classification()` function, so to get exactly the distribution as above, the used call was:

```
stream = StreamGenerator(
  n_classes=2,
  n_features=2,
  n_informative=2,
  n_redundant=0,
  n_repeated=0,
  n_features=2,
  random_state=105,
  n_chunks=100,
  chunk_size=500
)
```

What's very important, contrary to the typical call to `make_classification()`, we don't specify the `n_samples` parameter here, which determines the number of patterns in the set, but instead we provide two new attributes of data stream:

- `n_chunks` — to determine the number of chunks in a data stream.
- `chunk_size` — to determine the number of patterns in each data chunk.

Additionally, data streams may contain noise which, while not considered as concept drift, provides additional challenge during the data stream analysis and data stream classifiers should be robust to it. The `StreamGenerator` class implements noise by inverting the class labels of a given percentage of incoming instances in the data stream. This percentage can be defined by a `y_flip` parameter, like in standard `make_classification()` call. If a single float is given as the parameter value, the percentage of noise refers to combined instances from all classes, while if we specify a tuple of floats, the noise occurs within each class separately using the given percentages.

2.2 Streams containing concept drifts

The most commonly studied nature of data streams is their variability in time. Responsible for this is the phenomenon of the *concept drift*, where class distributions change over time with different dynamics, which necessitates the rebuilding of already fitted classification models. The `stream-learn` package tries to meet the need to synthesize all basic variations of this phenomenon (i.e. *sudden* (abrupt) and *gradual* drifts).

2.2.1 Sudden (Abrupt) drift

This type of drift occurs when the concept from which the data stream is generated is suddenly replaced by another one. Concept probabilities used by the `StreamGenerator` class are created based on sigmoid function, which is generated using `concept_sigmoid_spacing` parameter, which determines the function shape and how sudden the change of concept is. The higher the value, the more sudden the drift becames. Here, this parameter takes the default value of 999, which allows us for a generation of sigmoid function simulating an abrupt change in the data stream.

```
StreamGenerator(n_drifts=2)
```

2.2.2 Gradual drift

Unlike sudden drifts, gradual ones are associated with a slower change rate, which can be noticed during a longer observation of the data stream. This kind of drift refers to the transition phase where the probability of getting instances from the first concept decreases while the probability of sampling from the next concept increases. The `StreamGenerator` class simulates gradual drift by comparing the concept probabilities with the generated random noise and, depending on the result, selecting which concept is active at a given time.

```
StreamGenerator(
    n_drifts=2, concept_sigmoid_spacing=5
)
```

2.2.3 Incremental (Stepwise) drift

The incremental drift occurs when we are dealing with a series of barely noticeable changes in the concept used to generate the data stream, in opposite of gradual drift, where we are mixing samples from different concepts without changing them. Due to this, the drift may be identified only after some time. The severity of changes, and hence the speed of transition of one concept into another, is, like in previous example, described by the *concept_sigmoid_spacing* parameter.

```
StreamGenerator(
    n_drifts=2, concept_sigmoid_spacing=5, incremental=True
)
```

2.2.4 Recurrent drift

The cyclic repetition of class distributions is a completely different property of concept drifts. If after another drift, the concept earlier present in the stream returns, we are dealing with a *recurrent drift*. We can get this kind of data stream by setting the `recurring` flag in the generator.

```
StreamGenerator(
    n_drifts=2, recurring=True
)
```

2.2.5 Non-recurring drift

The default mode of consecutive concept occurences is a non-recurring drift, where in each concept drift we are generating a completely new, previously unseen class distribution.

```
StreamGenerator(
    n_drifts=2
)
```

2.3 Class imbalance

Another area of data stream properties, different from the concept drift phenomenon, is the prior probability of problem classes. By default, a balanced stream is generated, i.e. one in which patterns of all classes are present in a similar number.

```
StreamGenerator()
```

2.3.1 Stationary imbalanced stream

The basic type of problem in which we are dealing with disturbed class distribution is a *dataset imbalanced stationary*, where the classes maintain a predetermined proportion in each chunk of data stream. To acquire this type of a stream, one should pass the `list` to the `weights` parameter of the generator (i) consisting of as many elements as the classes in the problem and (ii) adding to one.

```
StreamGenerator(weights=[0.3, 0.7])
```

2.3.2 Dynamically imbalanced stream

A less common type of *imbalanced data*, impossible to obtain in static datasets, is *data imbalanced dynamically*. In this case, the class distribution is not constant throughout the course of a stream, but changes over time, similar to changing the concept presence in gradual streams. To get this type of data stream, we pass a `tuple` of three numeric values to the `weights` parameter of the generator:

- the number of cycles of distribution changes,
- `concept_sigmoid_spacing` parameter, deciding about the dynamics of changes on the same principle as in gradual and incremental drifts,
- range within which oscillation is to take place.

```
StreamGenerator(weights=(2, 5, 0.9))
```

2.4 Mixing drift properties

Of course, when generating data streams, we don't have to limit ourselves to just one modification of their properties. We can easily prepare a stream with many drifts, any dynamics of changes, a selected type of drift and a diverse, dynamic imbalanced ratio. The last example in this chapter of User Guide is such proposition, namely, DISCO (Dynamically Imbalanced Stream with Concept Oscillation).

```
StreamGenerator(
    weights=(2, 5, 0.9), n_drifts=3, concept_sigmoid_spacing=5,
    recurring=True, incremental=True
)
```

CHAPTER

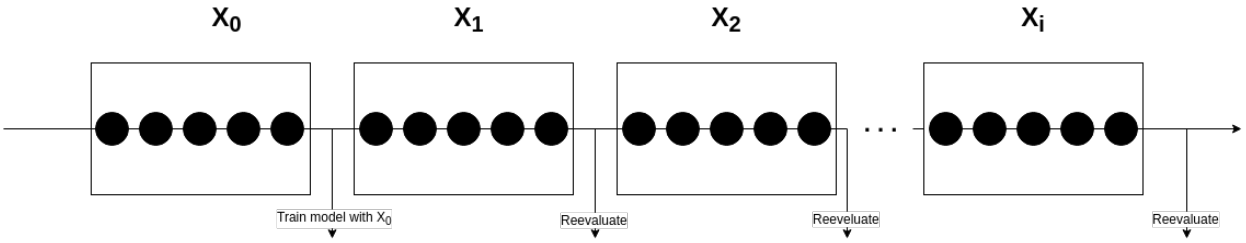
THREE

STREAM EVALUATORS

To estimate prediction measures in the context of data streams with strict computational requirements and concept drifts, the `evaluators` module of the `stream-learn` package implements two main estimation techniques described in the literature in their batch-based versions.

3.1 Test-Then-Train Evaluator

The `TestThenTrain` class implements the *Test-Then-Train* evaluation procedure, in which each individual data chunk is first used to test the classifier before it is used for updating the existing model.



The performance metrics returned by the evaluator are determined by the `metrics` parameter which accepts a tuple containing the functions of preferred quality measures and can be specified during initialization.

Processing of the data stream is started by calling the `process()` function which accepts two parameters (i.e. `stream` and `clfs`) responsible for defining the data stream and classifier, or a tuple of classifiers, employing the `partial_fit()` function. The size of each data chunk is determined by the `chunk_size` parameter from the `StreamGenerator` class. The results of evaluation can be accessed using the `scores` attribute, which is a three-dimensional array of shape (n_classifiers, n_chunks, n_metrics).

Example – single classifier

```
from strlearn.evaluators import TestThenTrain
from strlearn.ensembles import SEA
from strlearn.utils.metrics import bac, f_score
from strlearn.streams import StreamGenerator
from sklearn.naive_bayes import GaussianNB

stream = StreamGenerator(chunk_size=200, n_chunks=250)
clf = SEA(base_estimator=GaussianNB())
evaluator = TestThenTrain(metrics=(bac, f_score))

evaluator.process(stream, clf)
print(evaluator.scores)
```

Example – multiple classifiers

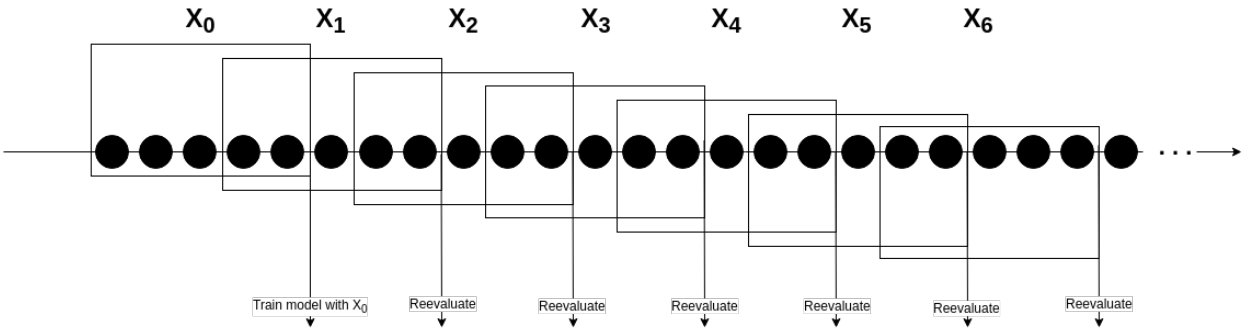
```
from strlearn.evaluators import TestThenTrain
from strlearn.ensembles import SEA
from strlearn.utils.metrics import bac, f_score
from strlearn.streams import StreamGenerator
from sklearn.naive_bayes import GaussianNB
from sklearn.tree import DecisionTreeClassifier

stream = StreamGenerator(chunk_size=200, n_chunks=250)
clf1 = SEA(base_estimator=GaussianNB())
clf2 = SEA(base_estimator=DecisionTreeClassifier())
clfs = (clf1, clf2)
evaluator = TestThenTrain(metrics=(bac, f_score))

evaluator.process(stream, clfs)
print(evaluator.scores)
```

3.2 Prequential Evaluator

The *Prequential* procedure of assessing the predictive performance of stream learning algorithms is implemented by the `Prequential` class. This estimation technique is based on a forgetting mechanism in the form of a sliding window instead of a separate data chunks. Window moves by a fixed number of instances determined by the `interval` parameter for the `process()` function. After each step, samples that are currently in the window are used to test the classifier and then for updating the model.



Similar to the `TestThenTrain` evaluator, the object of the `Prequential` class can be initialized with a `metrics` parameter containing metrics names and the size of the sliding window is equal to the `chunk_size` parameter from the instance of `StreamGenerator` class.

Example – single classifer

```
from strlearn.evaluators import Prequential
from strlearn.ensembles import SEA
from strlearn.utils.metrics import bac, f_score
from strlearn.streams import StreamGenerator
from sklearn.naive_bayes import GaussianNB

stream = StreamGenerator()
clf = SEA(base_estimator=GaussianNB())
evaluator = TestThenTrain(metrics=(bac, f_score))
```

(continues on next page)

(continued from previous page)

```
evaluator.process(stream, clf, interval=100)
print(evaluator.scores)
```

Example – multiple classifiers

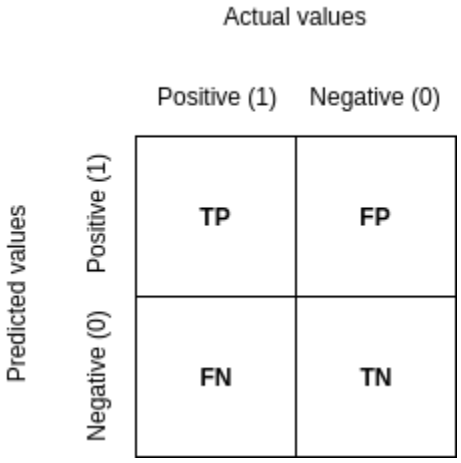
```
from strlearn.evaluators import Prequential
from strlearn.ensembles import SEA
from strlearn.utils.metrics import bac, f_score
from strlearn.streams import StreamGenerator
from sklearn.naive_bayes import GaussianNB
from sklearn.tree import DecisionTreeClassifier

stream = StreamGenerator(chunk_size=200, n_chunks=250)
clf1 = SEA(base_estimator=GaussianNB())
clf2 = SEA(base_estimator=DecisionTreeClassifier())
clfs = (clf1, clf2)
evaluator = Prequential(metrics=(bac, f_score))

evaluator.process(stream, clfs, interval=100)
print(evaluator.scores)
```

3.3 Metrics

To improve the computational performance of presented evaluators, the `stream-learn` package uses its own implementations of metrics for classification of imbalanced binary problems, which can be found in the `utils.metrics` module. All implemented metrics are based on the confusion matrix.



3.3.1 Recall

Recall (also known as sensitivity or true positive rate) represents the classifier's ability to find all the positive data samples in the dataset (e.g. the minority class instances) and is denoted as

$$Recall = \frac{tp}{tp + fn}$$

Example

```
from strlearn.utils.metrics import recall
```

3.3.2 Precision

Precision (also called positive predictive value) expresses the probability of correct detection of positive samples and is denoted as

$$Precision = \frac{tp}{tp + fp}$$

Example

```
from strlearn.utils.metrics import precision
```

3.3.3 F-beta score

The F-beta score can be interpreted as a weighted harmonic mean of precision and recall taking both metrics into account and punishing extreme values. The `beta` parameter determines the recall's weight. `beta` < 1 gives more weight to precision, while `beta` > 1 prefers recall. The formula for the F-beta score is

$$F_\beta = (1 + \beta^2) * \frac{Precision * Recall}{(\beta^2 * Precision) + Recall}$$

Example

```
from strlearn.utils.metrics import fbeta_score
```

3.3.4 F1 score

The F1 score can be interpreted as a F-beta score, where β parameter equals 1. It is a harmonic mean of precision and recall. The formula for the F1 score is

$$F_1 = 2 * \frac{Precision * Recall}{Precision + Recall}$$

Example

```
from strlearn.utils.metrics import f1_score
```

3.3.5 Balanced accuracy (BAC)

The balanced accuracy for the multiclass problems is defined as the average of recall obtained on each class. For binary problems it is denoted by the average of recall and specificity (also called true negative rate).

$$BAC = \frac{Recall + Specificity}{2}$$

Example

```
from strlearn.utils.metrics import bac
```

3.3.6 Geometric mean score 1 (G-mean1)

The geometric mean (G-mean) tries to maximize the accuracy on each of the classes while keeping these accuracies balanced. For N-class problems it is a N root of the product of class-wise recall. For binary classification G-mean is denoted as the squared root of the product of the recall and specificity.

$$Gmean1 = \sqrt{Recall * Specificity}$$

Example

```
from strlearn.utils.metrics import geometric_mean_score_1
```

3.3.7 Geometric mean score 2 (G-mean2)

The alternative definition of G-mean measure. For binary classification G-mean is denoted as the squared root of the product of the recall and precision.

$$Gmean2 = \sqrt{Recall * Precision}$$

Example

```
from strlearn.utils.metrics import geometric_mean_score_2
```

3.3.8 References

1. Ricardo A. Baeza-Yates and Berthier Ribeiro-Neto. *Modern Information Retrieval*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999. ISBN 020139829X.
2. Ricardo Barandela, Josep Sánchez, Vicente García, and E. Rangel. Strategies for learning in class imbalance problems. *Pattern Recognition*, 36:849–851, 03 2003.
3. Kay Henning Brodersen, Cheng Soon Ong, Klaas Enno Stephan, and Joachim M. Buhmann. The balanced accuracy and its posterior distribution. In *Proceedings of the 2010 20th International Conference on Pattern Recognition*, ICPR '10, 3121–3124. Washington, DC, USA, 2010. IEEE Computer Society.
4. Joao Gama. *Knowledge Discovery from Data Streams*. Chapman & Hall/CRC, 1st edition, 2010. ISBN 1439826110, 9781439826119.
5. João Gama, Raquel Sebastião, and Pedro Pereira Rodrigues. On evaluating stream learning algorithms. *Machine Learning*, 90(3):317–346, Mar 2013.

6. John D. Kelleher, Brian Mac Namee, and Aoife D'Arcy. *Fundamentals of Machine Learning for Predictive Data Analytics: Algorithms, Worked Examples, and Case Studies*. The MIT Press, 2015. ISBN 0262029448, 9780262029445.

7. Miroslav Kubat and Stan Matwin. Addressing the curse of imbalanced training sets: one-sided selection. In *ICML*. 1997.

8. David Powers and Ailab. Evaluation: from precision, recall and f-measure to roc, informedness, markedness & correlation. *J. Mach. Learn. Technol*, 2:2229–3981, 01 2011.

9. Yutaka Sasaki. The truth of the f-measure. *Teach Tutor Mater*, pages, 01 2007.

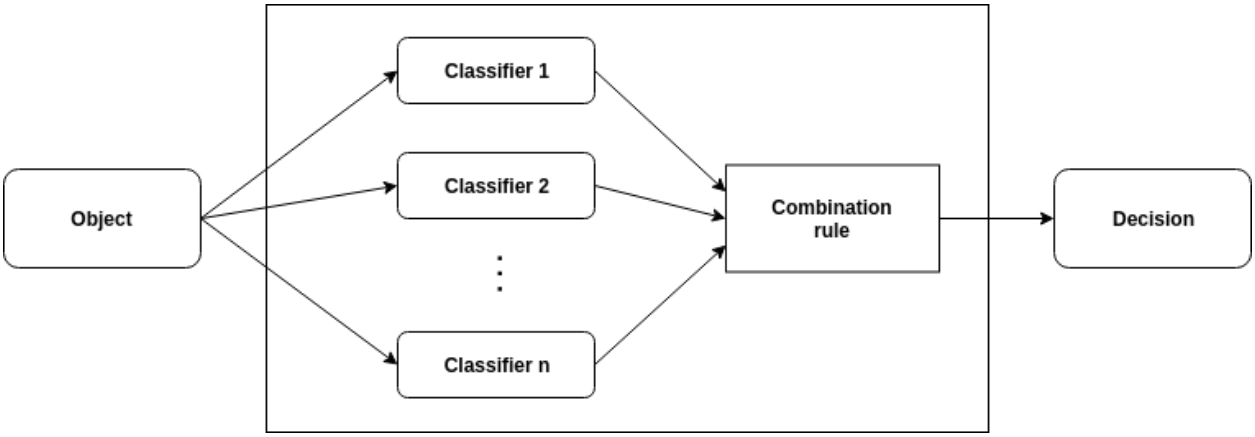
CHAPTER

FOUR

CLASSIFIER ENSEMBLES

An ensemble (also known as multiple classifier system or committee) consists of a set of base classifiers whose predictions are combined to label new instances. Combining classifiers have been proved to be an effective way of dividing complex learning problems into sub-problems as well as improving predictive accuracy. A well-tuned ensemble should contain both strong and diverse base models.

Classifier ensemble diagram



Under the data stream scenario, based on the way of examples processing, the ensembles can be categorized as *chunk-based* or *online*. The `stream-learn` package implements various ensemble methods for data stream classification, which can be found in the `ensembles` module.

4.1 Chunk-based Ensembles for Data Streams

Chunk-based approaches process successively incoming data chunks containing a predetermined number of instances. The learning algorithm can repeatedly process training samples located in a given data chunk to learn base models. It is worth noting that this does not mean that batch processing can only be used when new instances arrive in chunks. These approaches can also be used when instances arrive individually, if we store each new sample in a buffer until its size is equal to the size of the chunk.

4.1.1 Streaming Ensemble Algorithm (SEA)

The `SEA` class implements a basic multi classifier approach for data stream classification. This model takes the base classifier as the `base_estimator` parameter and the pool size as the `n_estimators`. A single base classifier is trained on each observed data chunk and added to the ensemble. If the fixed pool size is exceeded, the worst performing model is removed. The final decision is obtained by accumulating the supports of base classifiers.

Example

```
from strlearn.evaluators import TestThenTrain
from strlearn.streams import StreamGenerator
from strlearn.ensembles import SEA

from sklearn.naive_bayes import GaussianNB

stream = StreamGenerator()
clf = SEA(base_estimator=GaussianNB(), n_estimators=5)
evaluator = TestThenTrain()

evaluator.process(stream, clf)
print(evaluator.scores)
```

4.1.2 Weighted Aging Ensemble (WAE)

The `WAE` class implements an algorithm called Weighted Aging Ensemble, which can adapt to changes in data stream class distribution. The method was inspired by Accuracy Weighted Ensemble (AWE) algorithm to which it introduces two main modifications: (I) classifier weights depend on the individual classifier accuracies and time they have been spending in the ensemble, (II) individual classifier are chosen on the basis on the non-pairwise diversity measure. The `WAE` class accepts the following parameters:

- `base_estimator` – Base classifier type.
- `n_estimators` – Fixed pool size.
- `theta` – Threshold for weight calculation method and aging procedure control.
- `post_pruning` – Whether the pruning is conducted before or after adding the classifier.
- `pruning_criterion` – accuracy.
- `weight_calculation_method` – same_for_each, proportional_to_accuracy, kuncheva, pta_related_to_whole, bell_curve,
- `aging_method` – weights_proportional, constant, gaussian.
- `rejuvenation_power` – Rejuvenation dynamics control of classifiers with high prediction accuracy.

Example

```
from strlearn.evaluators import TestThenTrain
from strlearn.streams import StreamGenerator
from strlearn.ensembles import WAE

from sklearn.naive_bayes import GaussianNB

stream = StreamGenerator()
clf = sl.ensembles.WAE(
```

(continues on next page)

(continued from previous page)

```
      GaussianNB(), weight_calculation_method="proportional_to_accuracy"
  )
evaluator = TestThenTrain()

evaluator.process(stream, clf)
print(evaluator.scores)
```

4.2 Online Ensembles for Data Streams

Online approaches, unlike those based on batch processing, process each new sample separately. These methods have been developed for applications with memory and computational limitations (i.e. where the amount of incoming data is extensive). Online methods can also be used in cases where data samples do not arrive separately. These types of methods can process each instances of data chunk are individually and can therefore be used in an environment where data arrives in batches.

4.2.1 Online Bagging (OB)

Online Bagging is an ensemble learning algorithm for data streams classification, based on the concept of offline *Bagging*. It maintains a pool of base estimators and with the appearance of a new instance, each model is trained on it K times, where K comes from the *Poisson(= 1)* distribution. It is implemented in the `OnlineBagging` class which accepts `base_estimator` and `n_estimators` parameters, respectively responsible for the base classifier type and the fixed classifier pool size.

Example

```
from strlearn.evaluators import TestThenTrain
from strlearn.streams import StreamGenerator
from strlearn.ensembles import OnlineBagging

from sklearn.naive_bayes import GaussianNB

stream = StreamGenerator()
clf = OnlineBagging(base_estimator=GaussianNB(), n_estimators=5)
evaluator = TestThenTrain()

evaluator.process(stream, clf)
print(evaluator.scores)
```

4.2.2 Oversamping-Based Online Bagging (OOB) & Undersampling-Based Online Bagging (UOB)

Oversampling-Based Online Bagging (implemented by the `OOB` class) and *Undersampling-Based Online Bagging* (implemented by the `UOB` class) are methods integrating resampling with *Online Bagging*. Resampling is based on the change in values for the Poisson distribution. *OOB* uses oversampling to increase the chance of training minority class instances, while UOB uses undersampling to reduce the chance of training majority class instances. Implementations refer to the improved versions of both algorithms in which the value depends on the size ratio between classes. When the problem becomes balanced, the methods are automatically reduced to online bagging. Both methods take the same parameters as the `OnlineBagging` class.

Example

```
from strlearn.evaluators import TestThenTrain
from strlearn.streams import StreamGenerator
from strlearn.ensembles import OOB, UOB

from sklearn.naive_bayes import GaussianNB

stream = StreamGenerator()
oob = OOB(base_estimator=GaussianNB(), n_estimators=5)
uob = UOB(base_estimator=GaussianNB(), n_estimators=5)
clfs = (oob, uob)
evaluator = TestThenTrain()

evaluator.process(stream, clfs)
print(evaluator.scores)
```

4.2.3 References

1. Bartosz Krawczyk, Leandro Minku, João Gama, Jerzy Stefanowski, and Michal Wozniak. Ensemble learning for data stream analysis: a survey. *Information Fusion*, 37:132–156, 09 2017.
2. Nick Street and Y. Kim. A streaming ensemble algorithm (sea) for large-scale classification. *Proceedings of the 7Th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 377–382, 01 2001.
3. Michał Woźniak, Andrzej Kasprzak, and Piotr Cal. Weighted aging classifier ensemble for the incremental drifted data streams. In Henrik Legind Larsen, Maria J. Martin-Bautista, María Amparo Vila, Troels Andreasen, and Henning Christiansen, editors, *Flexible Query Answering Systems*, 579–588. Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.
4. N. C. Oza. Online bagging and boosting. In *2005 IEEE International Conference on Systems, Man and Cybernetics*, volume 3, 2340–2345 Vol. 3. Oct 2005.
5. S. Wang, L. L. Minku, and X. Yao. Resampling-based ensemble methods for online class imbalance learning. *IEEE Transactions on Knowledge and Data Engineering*, 27(5):1356–1368, May 2015.

CHAPTER

FIVE

CLASSIFIERS

In addition, the `stream-learn` library also implements a simple single classifier model implementing the `partial_fit()` method and a Meta estimator adapted to be used with some of the ensemble methods found in the `ensembles` module. Those two models can be found in the `classifiers` module.

5.1 Accumulated Samples Classifier

The `AccumulatedSamplesClassifier` class takes the base classifier as a `base_clf` parameter during initialization and extends the given model with the `partial_fit()` function adapted for data streams classification. This function concatenates observed data chunks, and in each step fits the model on all samples encountered so far.

Example

```
from strlearn.evaluators import TestThenTrain
from strlearn.streams import StreamGenerator
from strlearn.classifiers import AccumulatedSamplesClassifier

from sklearn.naive_bayes import GaussianNB

stream = StreamGenerator()
clf = AccumulatedSamplesClassifier(base_clf=GaussianNB())
evaluator = TestThenTrain()

evaluator.process(stream, clf)
print(evaluator.scores)
```

5.2 Sample-Weighted Meta Estimator

The `SampleWeightedMetaEstimator` class implements a meta estimator designed to allow the use of a wider range of classification models as base classifiers in ensemble methods based on *online bagging*. It extends the `partial_fit()` method of a given model by an additional `sample_weight` parameter which allows for using classifiers such as `MLPClassifier` from `scikit-learn` package as base models for `OnlineBagging`, `OOB` and `UOB` from `ensembles` module.

Example

```
from strlearn.evaluators import TestThenTrain
from strlearn.streams import StreamGenerator
from strlearn.classifiers import SampleWeightedMetaEstimator
```

(continues on next page)

(continued from previous page)

```
from strlearn.ensembles import OOB

from sklearn.neural_network import MLPClassifier


stream = StreamGenerator(n_chunks=10)
base = SampleWeightedMetaEstimator(base_classifier=MLPClassifier())
clf = OOB(base_estimator=base, n_estimators=2)
evaluator = TestThenTrain()

evaluator.process(stream, clf)
print(evaluator.scores)
```

CHAPTER

SIX

STREAMS MODULE

StreamGenerator([n_chunks, chunk_size, ...])	Data streams generator for both stationary and drifting data streams.
ARFFParser(path[, chunk_size, n_chunks])	Stream-aware parser of datasets in ARFF format.
CSVParser(path[, chunk_size, n_chunks])	Stream-aware parser of datasets in CSV format.
NPYParser(path[, chunk_size, n_chunks])	Stream-aware parser of datasets in numpy format.

class strlearn.streams.**ARFFParser**(*path*, *chunk_size=200*, *n_chunks=250*)

Bases: object

Stream-aware parser of datasets in ARFF format.

Parameters

- **path** (*string*) – Path to the ARFF file.
- **chunk_size** (*integer, optional (default=200)*) – The number of instances in each data chunk.
- **n_chunks** (*integer, optional (default=250)*) – The number of data chunks, that the stream is composed of.

Example

```
>>> import strlearn as sl
>>> stream = sl.streams.ARFFParser("Agrawal.arff")
>>> clf = sl.classifiers.AccumulatedSamplesClassifier()
>>> evaluator = sl.evaluators.PrequentialEvaluator()
>>> evaluator.process(clf, stream)
>>> stream.reset()
>>> print(evaluator.scores_)
...
[[0.855      0.80815508 0.79478582 0.80815508 0.89679715]
[0.795      0.75827674 0.7426779  0.75827674 0.84644195]
[0.8        0.75313899 0.73559983 0.75313899 0.85507246]
...
[0.885      0.86181169 0.85534199 0.86181169 0.91119691]
[0.895      0.86935764 0.86452058 0.86935764 0.92134831]
[0.87       0.85104088 0.84813907 0.85104088 0.9       ]]
```

get_chunk()

Generating a data chunk of a stream.

Used by all evaluators but also accesible for custom evaluation.

Returns
Generated samples and target values.

Return type
tuple {array-like, shape (n_samples, n_features), array-like, shape (n_samples,)}

`is_dry()`

Checking if we have reached the end of the stream.

Returns
flag showing if the stream has ended

Return type
boolean

`reset()`

Reset processed stream and close ARFF file.

`class strlearn.streams.CSVParser(path, chunk_size=200, n_chunks=250)`

Bases: `object`

Stream-aware parser of datasets in CSV format.

Parameters

- **path** (*string*) – Path to the csv file.
- **chunk_size** (*integer, optional (default=200)*) – The number of instances in each data chunk.
- **n_chunks** (*integer, optional (default=250)*) – The number of data chunks, that the stream is composed of.

Example

```
>>> import strlearn as sl
>>> stream = sl.streams.CSVParser("Agrawal.csv")
>>> clf = sl.classifiers.AccumulatedSamplesClassifier()
>>> evaluator = sl.evaluators.PrequentialEvaluator()
>>> evaluator.process(clf, stream)
>>> stream.reset()
>>> print(evaluator.scores_)
...
[[0.855      0.80815508 0.79478582 0.80815508 0.89679715]
[0.795      0.75827674 0.7426779  0.75827674 0.84644195]
[0.8        0.75313899 0.73559983 0.75313899 0.85507246]
...
[0.885      0.86181169 0.85534199 0.86181169 0.91119691]
[0.895      0.86935764 0.86452058 0.86935764 0.92134831]
[0.87       0.85104088 0.84813907 0.85104088 0.9       ]]
```

`get_chunk()`

Generating a data chunk of a stream.

Used by all evaluators but also accesible for custom evaluation.

Returns
Generated samples and target values.

Return type
tuple {array-like, shape (n_samples, n_features), array-like, shape (n_samples,)}

is_dry()

Checking if we have reached the end of the stream.

Returns
flag showing if the stream has ended

Return type
boolean

reset()

Reset stream to the beginning.

class strlearn.streams.**NPYParser**(*path*, *chunk_size=200*, *n_chunks=250*)

Bases: object

Stream-aware parser of datasets in numpy format.

Parameters

- **path** (*string*) – Path to the npy file.
- **chunk_size** (*integer, optional (default=200)*) – The number of instances in each data chunk.
- **n_chunks** (*integer, optional (default=250)*) – The number of data chunks, that the stream is composed of.

Example

```
>>> import strlearn as sl
>>> stream = sl.streams.NPYParser("Agrawal.npy")
>>> clf = sl.classifiers.AccumulatedSamplesClassifier()
>>> evaluator = sl.evaluators.PrequentialEvaluator()
>>> evaluator.process(clf, stream)
>>> stream.reset()
>>> print(evaluator.scores_)
...
[[0.855      0.80815508 0.79478582 0.80815508 0.89679715]
[0.795      0.75827674 0.7426779  0.75827674 0.84644195]
[0.8        0.75313899 0.73559983 0.75313899 0.85507246]
...
[0.885      0.86181169 0.85534199 0.86181169 0.91119691]
[0.895      0.86935764 0.86452058 0.86935764 0.92134831]
[0.87       0.85104088 0.84813907 0.85104088 0.9       ]]
```

get_chunk()

Generating a data chunk of a stream.

Used by all evaluators but also accesible for custom evaluation.

Returns
Generated samples and target values.

Return type
tuple {array-like, shape (n_samples, n_features), array-like, shape (n_samples,)}

is_dry()

Checking if we have reached the end of the stream.

Returns
flag showing if the stream has ended

Return type
boolean

reset()

Reset stream to the beginning.

class strlearn.streams.**SemiSyntheticStreamGenerator**(*X*, *y*, *n_chunks=200*, *chunk_size=250*, *random_state=None*, *n_drifts=2*, *n_features=10*, *interpolation='nearest'*, *stabilize_factor=0.2*, *binarize=True*)

Bases: object

Semi-Synthetic Data streams generator for drifting data streams.

A generator that allows preparing a replicable classification dataset based on real-world input data. The generator uses one-dimensional interpolation to generate the drifting projections, based on which the final data stream is generated.

Parameters

- **X** (*array-like, shape (n_samples, n_features)*) – Static dataset features.
- **y** (*array-like, shape (n_samples,)*) – Static dataset labels.
- **n_chunks** (*integer, optional (default=200)*) – The number of data chunks, that the stream is composed of.
- **chunk_size** (*integer, optional (default=250)*) – The number of instances in each data chunk.
- **random_state** (*integer, optional (default=None)*) – The seed used by the random number generator.
- **n_drifts** (*integer, optional (default=2)*) – The number of concept changes in the data stream.
- **n_features** (*integer, optional (default=10)*) – The number of features in output stream.
- **interpolation** (*string, optional (default='nearest')*) – Interpolation type.
- **stabilize_factor** (*float, optional (default=0.2)*) – The factor describing the stability of a concept.
- **binarize** (*boolean, optional (default=True)*) – Flag describing if the data should be binarized.

Example

```
>>> import strlearn as sl
>>> from sklearn.datasets import load_breast_cancer
>>> from sklearn.naive_bayes import GaussianNB
>>> X, y =  load_breast_cancer(return_X_y=True)
>>> stream = sl.streams.SemiSyntheticStreamGenerator(X, y, n_drifts=4,␣
↪interpolation='cubic')
>>> clf = GaussianNB()
>>> evaluator = sl.evaluators.TestThenTrain()
>>> evaluator.process(stream, clf)
>>> print(stream._get_drifts())
[ 14  48  89 155]
```

get_chunk()

Generating a data chunk of a stream.

Used by all evaluators but also accesible for custom evaluation.

Returns
Generated samples and target values.

Return type
tuple {array-like, shape (n_samples, n_features), array-like, shape (n_samples,)}

save_to_arff(*filepath*)

Save generated stream to the ARFF format file.

Parameters
filepath (*string*) – Path to the file where data will be saved in ARFF format.

save_to_csv(*filepath*)

Save generated stream to the csv format file.

Parameters
filepath (*string*) – Path to the file where data will be saved in csv format.

save_to_npy(*filepath*)

Save generated stream to the numpy format file.

Parameters
filepath (*string*) – Path to the file where data will be saved in numpy format.

class strlearn.streams.**StreamGenerator**(*n_chunks=250*, *chunk_size=200*, *random_state=None*, *n_drifts=0*, *concept_sigmoid_spacing=None*, *n_classes=2*, *n_features=20*, *n_informative=2*, *n_redundant=2*, *n_repeated=0*, *n_clusters_per_class=2*, *recurring=False*, *weights=None*, *incremental=False*, *y_flip=0.01*, ***kwargs*)

Bases: object

Data streams generator for both stationary and drifting data streams.

A key element of the stream-learn package is a generator that allows to prepare a replicable (according to the given random_state value) classification dataset with class distribution changing over the course of stream, with base concepts build on a default class distributions for the scikit-learn package from the make_classification() function. These types of distributions try to reproduce the rules for generating the Madelon set. The StreamGenerator is capable of preparing any variation of the data stream known in the general taxonomy of data streams.

Parameters

- **n_chunks** (*integer, optional (default=250)*) – The number of data chunks, that the stream is composed of.
- **chunk_size** (*integer, optional (default=200)*) – The number of instances in each data chunk.
- **random_state** (*integer, optional (default=1410)*) – The seed used by the random number generator.
- **n_drifts** (*integer, optional (default=4)*) – The number of concept changes in the data stream.
- **concept_sigmoid_spacing** (*float, optional (default=10.)*) – Value that determines the shape of sigmoid function and how sudden is the change of concept. The higher the value, the more sudden the drift is.

- **n_classes** (*integer, optional (default=2)*) – The number of classes in the generated data stream.
- **y_flip** (*float or tuple (default=0.01)*) – Label noise for whole dataset or separate classes.
- **recurring** (*boolean, optional (default=False)*) – Determines if the streams can go back to the previously encountered concepts.
- **weights** (*array-like, shape (n_classes,) or tuple (only for 2 classes)*) – If array - class weight for static imbalance, if 3-valued tuple - (n_drifts, concept_sigmoid_spacing, IR amplitude [0-1]) for generation of continous dynamically imbalanced streams, if 2-valued tuple - (mean value, standard deviation) for generation of discreete dynamically imbalanced streams.

Example

```
>>> import strlearn as sl
>>> stream = sl.streams.StreamGenerator(n_drifts=2, weights=[0.2, 0.8], concept_
↪sigmoid_spacing=5)
>>> clf = sl.classifiers.AccumulatedSamplesClassifier()
>>> evaluator = sl.evaluators.PrequentialEvaluator()
>>> evaluator.process(clf, stream)
>>> print(evaluator.scores_)
[[0.955      0.93655817 0.93601827 0.93655817 0.97142857]
 [0.94       0.91397849 0.91275313 0.91397849 0.96129032]
 [0.9        0.85565271 0.85234488 0.85565271 0.93670886]
 ...
 [0.815      0.72584133 0.70447376 0.72584133 0.8802589 ]
 [0.83       0.69522145 0.65223303 0.69522145 0.89570552]
 [0.845      0.67267706 0.61257135 0.67267706 0.90855457]]
```

get_chunk()

Generating a data chunk of a stream.

Used by all evaluators but also accesible for custom evaluation.

Returns
Generated samples and target values.

Return type
tuple {array-like, shape (n_samples, n_features), array-like, shape (n_samples,)}

save_to_arff(*filepath*)

Save generated stream to the ARFF format file.

Parameters
filepath (*string*) – Path to the file where data will be saved in ARFF format.

save_to_csv(*filepath*)

Save generated stream to the csv format file.

Parameters
filepath (*string*) – Path to the file where data will be saved in csv format.

save_to_npy(*filepath*)

Save generated stream to the numpy format file.

Parameters
filepath (*string*) – Path to the file where data will be saved in numpy format.

CHAPTER

SEVEN

EVALUATORS MODULE

Prequential([metrics])	Prequential data stream evaluator.
TestThenTrain([metrics, verbose])	Test Than Train data stream evaluator.

class strlearn.evaluators.**Prequential**(*metrics=(<function accuracy_score>, <function balanced_accuracy_score>)*)

Bases: object

Prequential data stream evaluator.

Implementation of prequential evaluation procedure, based on sliding windows instead of separate data chunks. Window moves by a fixed number of instances in order to preserve some of the already processed ones. After each step, samples that are currently in the window are used to test the classifier and then for training.

Parameters

metrics (*tuple or function*) – Tuple of metric functions or single metric function.

Variables

- **classes** (*array-like, shape (n_classes,)*) – The class labels.
- **scores** (*array-like, shape (stream.n_chunks, len(metrics))*) – Values of metrics for each processed data chunk.

Example

```
>>> import strlearn as sl
>>> stream = sl.streams.StreamGenerator()
>>> clf = sl.classifiers.AccumulatedSamplesClassifier()
>>> evaluator = sl.evaluators.PrequentialEvaluator()
>>> evaluator.process(clf, stream, interval=50)
>>> print(evaluator.scores_)
...
[[0.95       0.9483469  0.94805282 0.9483469  0.95412844]
[0.96       0.95728313 0.95696445 0.95728313 0.96460177]
[0.96       0.95858586 0.95848154 0.95858586 0.96396396]
...
[0.92       0.91987179 0.91986621 0.91987179 0.91666667]
[0.91       0.91065705 0.91050889 0.91065705 0.90816327]
[0.925      0.92567027 0.9250634  0.92567027 0.92610837]]
```

process(*stream*, *clfs*, *interval=100*)

Perform learning procedure on data stream.

Parameters

- **stream** (*object*) – Data stream as an object
- **clfs** (*tuple or function*) – scikit-learn estimator of list of scikit-learn estimators.
- **interval** (*integer, optional (default=100)*) – The number of instances by which the sliding window moves before the next evaluation and training steps.

class strlearn.evaluators.**SparseTrainDenseTest**(*n_repeats=5*, *metrics=(<function accuracy_score>*, *<function balanced_accuracy_score>)*, *verbose=False*)

Bases: object

Sparse Train Dense Test data stream evaluator.

Implementation of sparse-train-dense-test evaluation procedure, where each individual data chunk is first used to test the classifier and then it is used for given number of chunks of training.

Parameters

- **metrics** (*tuple or function*) – Tuple of metric functions or single metric function.
- **verbose** (*boolean*) – Flag to turn on verbose mode.

Variables

- **classes** (*array-like, shape (n_classes,)*) – The class labels.
- **scores** (*array-like, shape (stream.n_chunks, len(metrics))*) – Values of metrics for each processed data chunk.

process(*stream*, *clfs*)

Perform learning procedure on data stream.

Parameters

- **stream** (*object*) – Data stream as an object
- **clfs** (*tuple or function*) – scikit-learn estimator of list of scikit-learn estimators.

class strlearn.evaluators.**TestThenTrain**(*metrics=(<function accuracy_score>*, *<function balanced_accuracy_score>)*, *verbose=False*)

Bases: object

Test Than Train data stream evaluator.

Implementation of test-then-train evaluation procedure, where each individual data chunk is first used to test the classifier and then it is used for training.

Parameters

- **metrics** (*tuple or function*) – Tuple of metric functions or single metric function.
- **verbose** (*boolean*) – Flag to turn on verbose mode.

Variables

- **classes** (*array-like, shape (n_classes,)*) – The class labels.
- **scores** (*array-like, shape (stream.n_chunks, len(metrics))*) – Values of metrics for each processed data chunk.

Example

```
>>> import strlearn as sl
>>> stream = sl.streams.StreamGenerator()
>>> clf = sl.classifiers.AccumulatedSamplesClassifier()
>>> evaluator = sl.evaluators.TestThenTrainEvaluator()
>>> evaluator.process(clf, stream)
>>> print(evaluator.scores_)
...
[[0.92       0.91879699 0.91848191 0.91879699 0.92523364]
[0.945      0.94648779 0.94624912 0.94648779 0.94240838]
[0.92       0.91936979 0.91936231 0.91936979 0.9047619 ]
...
[0.92       0.91907051 0.91877671 0.91907051 0.9245283 ]
[0.885      0.8854889  0.88546135 0.8854889  0.87830688]
[0.935      0.93569212 0.93540766 0.93569212 0.93467337]]
```

process(*stream*, *clfs*)

Perform learning procedure on data stream.

Parameters

- **stream** (*object*) – Data stream as an object
- **clfs** (*tuple or function*) – scikit-learn estimator of list of scikit-learn estimators.

CHAPTER

EIGHT

ENSEMBLES MODULE

StreamingEnsemble(base_estimator, n_estimators)	Abstract, base ensemble streaming class
AUE([base_estimator, n_estimators, ...])	Accuracy Updated Ensemble
AWE([base_estimator, n_estimators, n_splits])	Accuracy Weighted Ensemble
DWM([base_estimator, beta, theta, p, weighted])	
KMC([base_estimator, n_estimators])	Wang, Yi, Yang Zhang, and Yong Wang. "Mining data streams
CDS([base_estimator, n_estimators, a, b])	Ditzler, Gregory, and Robi Polikar.
NIE([base_estimator, n_estimators, param_a, ...])	Ditzler, Gregory, and Robi Polikar.
OnlineBagging([base_estimator, n_estimators])	Online Bagging.
OOB([base_estimator, n_estimators, ...])	Oversamping-Based Online Bagging.
OUSE([base_estimator, n_estimators, n_chunks])	Gao, Jing, et al. "Classifying Data Streams with Skewed Class Distributions and Concept Drifts." IEEE Internet Computing 12.6 (2008): 37-49.
REA([base_estimator, n_estimators, ...])	Recursive Ensemble Approach.
SEA([base_estimator, n_estimators, metric])	Streaming Ensemble Algorithm.
UOB([base_estimator, n_estimators, ...])	Undersampling-Based Online Bagging.
WAE([base_estimator, n_estimators, theta, ...])	Weighted Aging Ensemble.
KUE([base_estimator, n_estimators, n_candidates])	Kappa Updated Ensemble

class strlearn.ensembles.**AUE**(*base_estimator=None*, *n_estimators=10*, *n_splits=5*, *epsilon=1e-10*)

Bases: *StreamingEnsemble*

Accuracy Updated Ensemble

partial_fit(*X*, *y*, *classes=None*)

Partial fitting

set_partial_fit_request(*, *classes: bool | None | str = '$UNCHANGED$'*) → *AUE*

Request metadata passed to the `partial_fit` method.

Note that this method is only relevant if `enable_metadata_routing=True` (see `sklearn.set_config()`). Please see User Guide on how the routing mechanism works.

The options for each parameter are:

- `True`: metadata is requested, and passed to `partial_fit` if provided. The request is ignored if metadata is not provided.
- `False`: metadata is not requested and the meta-estimator will not pass it to `partial_fit`.
- `None`: metadata is not requested, and the meta-estimator will raise an error if the user provides it.

- `str`: metadata should be passed to the meta-estimator with this given alias instead of the original name.

The default (`sklearn.utils.metadata_routing.UNCHANGED`) retains the existing request. This allows you to change the request for some parameters and not others.

Added in version 1.3.

Note: This method is only relevant if this estimator is used as a sub-estimator of a meta-estimator, e.g. used inside a `Pipeline`. Otherwise it has no effect.

8.1 Parameters

classes
[str, True, False, or None, default=sklearn.utils.metadata_routing.UNCHANGED] Metadata routing for `classes` parameter in `partial_fit`.

8.2 Returns

self
[object] The updated object.

`set_score_request`(*, *sample_weight: bool | None | str = '$UNCHANGED$'*) → *AUE*

Request metadata passed to the `score` method.

Note that this method is only relevant if `enable_metadata_routing=True` (see `sklearn.set_config()`). Please see User Guide on how the routing mechanism works.

The options for each parameter are:

- `True`: metadata is requested, and passed to `score` if provided. The request is ignored if metadata is not provided.
- `False`: metadata is not requested and the meta-estimator will not pass it to `score`.
- `None`: metadata is not requested, and the meta-estimator will raise an error if the user provides it.
- `str`: metadata should be passed to the meta-estimator with this given alias instead of the original name.

The default (`sklearn.utils.metadata_routing.UNCHANGED`) retains the existing request. This allows you to change the request for some parameters and not others.

Added in version 1.3.

Note: This method is only relevant if this estimator is used as a sub-estimator of a meta-estimator, e.g. used inside a `Pipeline`. Otherwise it has no effect.

8.3 Parameters

sample_weight
[str, True, False, or None, default=sklearn.utils.metadata_routing.UNCHANGED] Metadata routing for `sample_weight` parameter in `score`.

8.4 Returns

self
[object] The updated object.

class strlearn.ensembles.**AWE**(*base_estimator=None*, *n_estimators=10*, *n_splits=5*)

Bases: *StreamingEnsemble*

Accuracy Weighted Ensemble

partial_fit(*X*, *y*, *classes=None*)

Partial fitting

set_partial_fit_request(*, *classes: bool | None | str = '$UNCHANGED$'*) → *AWE*

Request metadata passed to the `partial_fit` method.

Note that this method is only relevant if `enable_metadata_routing=True` (see `sklearn.set_config()`). Please see User Guide on how the routing mechanism works.

The options for each parameter are:

- `True`: metadata is requested, and passed to `partial_fit` if provided. The request is ignored if metadata is not provided.
- `False`: metadata is not requested and the meta-estimator will not pass it to `partial_fit`.
- `None`: metadata is not requested, and the meta-estimator will raise an error if the user provides it.
- `str`: metadata should be passed to the meta-estimator with this given alias instead of the original name.

The default (`sklearn.utils.metadata_routing.UNCHANGED`) retains the existing request. This allows you to change the request for some parameters and not others.

Added in version 1.3.

Note: This method is only relevant if this estimator is used as a sub-estimator of a meta-estimator, e.g. used inside a `Pipeline`. Otherwise it has no effect.

8.5 Parameters

classes
[str, True, False, or None, default=sklearn.utils.metadata_routing.UNCHANGED] Metadata routing for `classes` parameter in `partial_fit`.

8.6 Returns

self
[object] The updated object.

`set_score_request`(*, *sample_weight: bool | None | str = '$UNCHANGED$'*) → *AWE*

Request metadata passed to the `score` method.

Note that this method is only relevant if `enable_metadata_routing=True` (see `sklearn.set_config()`). Please see User Guide on how the routing mechanism works.

The options for each parameter are:

- `True`: metadata is requested, and passed to `score` if provided. The request is ignored if metadata is not provided.
- `False`: metadata is not requested and the meta-estimator will not pass it to `score`.
- `None`: metadata is not requested, and the meta-estimator will raise an error if the user provides it.
- `str`: metadata should be passed to the meta-estimator with this given alias instead of the original name.

The default (`sklearn.utils.metadata_routing.UNCHANGED`) retains the existing request. This allows you to change the request for some parameters and not others.

Added in version 1.3.

Note: This method is only relevant if this estimator is used as a sub-estimator of a meta-estimator, e.g. used inside a `Pipeline`. Otherwise it has no effect.

8.7 Parameters

sample_weight
[str, True, False, or None, default=sklearn.utils.metadata_routing.UNCHANGED] Metadata routing for `sample_weight` parameter in `score`.

8.8 Returns

self
[object] The updated object.

class strlearn.ensembles.**CDS**(*base_estimator=None*, *n_estimators=10*, *a=2*, *b=2*)

Bases: *StreamingEnsemble*

Ditzler, Gregory, and Robi Polikar. "Incremental learning of concept drift from streaming imbalanced data." IEEE Transactions on Knowledge and Data Engineering 25.10 (2013): 2283-2301.

partial_fit(*X*, *y*, *classes=None*)

Partial fitting

set_partial_fit_request(*, *classes: bool | None | str = '$UNCHANGED$'*) → *CDS*

Request metadata passed to the `partial_fit` method.

Note that this method is only relevant if `enable_metadata_routing=True` (see `sklearn.set_config()`). Please see User Guide on how the routing mechanism works.

The options for each parameter are:

- `True`: metadata is requested, and passed to `partial_fit` if provided. The request is ignored if metadata is not provided.
- `False`: metadata is not requested and the meta-estimator will not pass it to `partial_fit`.
- `None`: metadata is not requested, and the meta-estimator will raise an error if the user provides it.
- `str`: metadata should be passed to the meta-estimator with this given alias instead of the original name.

The default (`sklearn.utils.metadata_routing.UNCHANGED`) retains the existing request. This allows you to change the request for some parameters and not others.

Added in version 1.3.

Note: This method is only relevant if this estimator is used as a sub-estimator of a meta-estimator, e.g. used inside a `Pipeline`. Otherwise it has no effect.

8.9 Parameters

classes
[str, True, False, or None, default=sklearn.utils.metadata_routing.UNCHANGED] Metadata routing for `classes` parameter in `partial_fit`.

8.10 Returns

self
[object] The updated object.

`set_score_request`(*, *sample_weight: bool | None | str = '$UNCHANGED$'*) → *CDS*

Request metadata passed to the `score` method.

Note that this method is only relevant if `enable_metadata_routing=True` (see `sklearn.set_config()`). Please see User Guide on how the routing mechanism works.

The options for each parameter are:

- `True`: metadata is requested, and passed to `score` if provided. The request is ignored if metadata is not provided.
- `False`: metadata is not requested and the meta-estimator will not pass it to `score`.
- `None`: metadata is not requested, and the meta-estimator will raise an error if the user provides it.
- `str`: metadata should be passed to the meta-estimator with this given alias instead of the original name.

The default (`sklearn.utils.metadata_routing.UNCHANGED`) retains the existing request. This allows you to change the request for some parameters and not others.

Added in version 1.3.

Note: This method is only relevant if this estimator is used as a sub-estimator of a meta-estimator, e.g. used inside a `Pipeline`. Otherwise it has no effect.

8.11 Parameters

sample_weight
[str, True, False, or None, default=sklearn.utils.metadata_routing.UNCHANGED] Metadata routing for `sample_weight` parameter in `score`.

8.12 Returns

self
[object] The updated object.

class `strlearn.ensembles.DWM`(*base_estimator=None*, *beta=0.5*, *theta=0.01*, *p=1*, *weighted=False*)

Bases: *StreamingEnsemble*

`partial_fit`(*X*, *y*, *classes=None*)

Partial fitting

`set_partial_fit_request`(*, *classes: bool | None | str = '$UNCHANGED$'*) → *DWM*

Request metadata passed to the `partial_fit` method.

Note that this method is only relevant if `enable_metadata_routing=True` (see `sklearn.set_config()`). Please see User Guide on how the routing mechanism works.

The options for each parameter are:

- `True`: metadata is requested, and passed to `partial_fit` if provided. The request is ignored if metadata is not provided.
- `False`: metadata is not requested and the meta-estimator will not pass it to `partial_fit`.
- `None`: metadata is not requested, and the meta-estimator will raise an error if the user provides it.
- `str`: metadata should be passed to the meta-estimator with this given alias instead of the original name.

The default (`sklearn.utils.metadata_routing.UNCHANGED`) retains the existing request. This allows you to change the request for some parameters and not others.

Added in version 1.3.

Note: This method is only relevant if this estimator is used as a sub-estimator of a meta-estimator, e.g. used inside a `Pipeline`. Otherwise it has no effect.

8.13 Parameters

classes
: [str, True, False, or None, default=sklearn.utils.metadata_routing.UNCHANGED] Metadata routing for `classes` parameter in `partial_fit`.

8.14 Returns

self
: [object] The updated object.

`set_score_request`(*, *sample_weight: bool | None | str = '$UNCHANGED$'*) → *DWM*

Request metadata passed to the `score` method.

Note that this method is only relevant if `enable_metadata_routing=True` (see `sklearn.set_config()`). Please see User Guide on how the routing mechanism works.

The options for each parameter are:

- `True`: metadata is requested, and passed to `score` if provided. The request is ignored if metadata is not provided.
- `False`: metadata is not requested and the meta-estimator will not pass it to `score`.
- `None`: metadata is not requested, and the meta-estimator will raise an error if the user provides it.
- `str`: metadata should be passed to the meta-estimator with this given alias instead of the original name.

The default (`sklearn.utils.metadata_routing.UNCHANGED`) retains the existing request. This allows you to change the request for some parameters and not others.

Added in version 1.3.

Note: This method is only relevant if this estimator is used as a sub-estimator of a meta-estimator, e.g. used inside a `Pipeline`. Otherwise it has no effect.

8.15 Parameters

sample_weight
[str, True, False, or None, default=sklearn.utils.metadata_routing.UNCHANGED] Metadata routing for `sample_weight` parameter in `score`.

8.16 Returns

self
[object] The updated object.

`class strlearn.ensembles.KMC(base_estimator=None, n_estimators=10)`

Bases: *StreamingEnsemble*

Wang, Yi, Yang Zhang, and Yong Wang. "Mining data streams
with skewed distribution by static classifier ensemble." Opportunities and Challenges for Next-Generation Applied Intelligence. Springer, Berlin, Heidelberg, 2009. 65-71.

`partial_fit(X, y, classes=None)`

Partial fitting

`set_partial_fit_request(*, classes: bool | None | str = '$UNCHANGED$') → KMC`

Request metadata passed to the `partial_fit` method.

Note that this method is only relevant if `enable_metadata_routing=True` (see `sklearn.set_config()`). Please see User Guide on how the routing mechanism works.

The options for each parameter are:

- `True`: metadata is requested, and passed to `partial_fit` if provided. The request is ignored if metadata is not provided.
- `False`: metadata is not requested and the meta-estimator will not pass it to `partial_fit`.
- `None`: metadata is not requested, and the meta-estimator will raise an error if the user provides it.
- `str`: metadata should be passed to the meta-estimator with this given alias instead of the original name.

The default (`sklearn.utils.metadata_routing.UNCHANGED`) retains the existing request. This allows you to change the request for some parameters and not others.

Added in version 1.3.

Note: This method is only relevant if this estimator is used as a sub-estimator of a meta-estimator, e.g. used inside a `Pipeline`. Otherwise it has no effect.

8.17 Parameters

classes
: [str, True, False, or None, default=sklearn.utils.metadata_routing.UNCHANGED] Metadata routing for `classes` parameter in `partial_fit`.

8.18 Returns

self
: [object] The updated object.

`set_score_request`(*, *sample_weight: bool | None | str = '$UNCHANGED$'*) → *KMC*

Request metadata passed to the `score` method.

Note that this method is only relevant if `enable_metadata_routing=True` (see `sklearn.set_config()`). Please see User Guide on how the routing mechanism works.

The options for each parameter are:

- `True`: metadata is requested, and passed to `score` if provided. The request is ignored if metadata is not provided.
- `False`: metadata is not requested and the meta-estimator will not pass it to `score`.
- `None`: metadata is not requested, and the meta-estimator will raise an error if the user provides it.
- `str`: metadata should be passed to the meta-estimator with this given alias instead of the original name.

The default (`sklearn.utils.metadata_routing.UNCHANGED`) retains the existing request. This allows you to change the request for some parameters and not others.

Added in version 1.3.

Note: This method is only relevant if this estimator is used as a sub-estimator of a meta-estimator, e.g. used inside a `Pipeline`. Otherwise it has no effect.

8.19 Parameters

sample_weight
: [str, True, False, or None, default=sklearn.utils.metadata_routing.UNCHANGED] Metadata routing for `sample_weight` parameter in `score`.

8.20 Returns

self
[object] The updated object.

class strlearn.ensembles.**KUE**(*base_estimator=None*, *n_estimators=10*, *n_candidates=1*)

Bases: *StreamingEnsemble*

Kappa Updated Ensemble

ensemble_support_matrix(*X*)

Ensemble support matrix.

partial_fit(*X*, *y*, *classes=None*)

Partial fitting

set_partial_fit_request(*, *classes: bool | None | str = '$UNCHANGED$'*) → *KUE*

Request metadata passed to the `partial_fit` method.

Note that this method is only relevant if `enable_metadata_routing=True` (see `sklearn.set_config()`). Please see User Guide on how the routing mechanism works.

The options for each parameter are:

- `True`: metadata is requested, and passed to `partial_fit` if provided. The request is ignored if metadata is not provided.
- `False`: metadata is not requested and the meta-estimator will not pass it to `partial_fit`.
- `None`: metadata is not requested, and the meta-estimator will raise an error if the user provides it.
- `str`: metadata should be passed to the meta-estimator with this given alias instead of the original name.

The default (`sklearn.utils.metadata_routing.UNCHANGED`) retains the existing request. This allows you to change the request for some parameters and not others.

Added in version 1.3.

Note: This method is only relevant if this estimator is used as a sub-estimator of a meta-estimator, e.g. used inside a `Pipeline`. Otherwise it has no effect.

8.21 Parameters

classes
[str, True, False, or None, default=sklearn.utils.metadata_routing.UNCHANGED] Metadata routing for `classes` parameter in `partial_fit`.

8.22 Returns

self
[object] The updated object.

`set_score_request`(*, *sample_weight: bool | None | str = '$UNCHANGED$'*) → *KUE*

Request metadata passed to the `score` method.

Note that this method is only relevant if `enable_metadata_routing=True` (see `sklearn.set_config()`). Please see User Guide on how the routing mechanism works.

The options for each parameter are:

- `True`: metadata is requested, and passed to `score` if provided. The request is ignored if metadata is not provided.
- `False`: metadata is not requested and the meta-estimator will not pass it to `score`.
- `None`: metadata is not requested, and the meta-estimator will raise an error if the user provides it.
- `str`: metadata should be passed to the meta-estimator with this given alias instead of the original name.

The default (`sklearn.utils.metadata_routing.UNCHANGED`) retains the existing request. This allows you to change the request for some parameters and not others.

Added in version 1.3.

Note: This method is only relevant if this estimator is used as a sub-estimator of a meta-estimator, e.g. used inside a `Pipeline`. Otherwise it has no effect.

8.23 Parameters

sample_weight
[str, True, False, or None, default=sklearn.utils.metadata_routing.UNCHANGED] Metadata routing for `sample_weight` parameter in `score`.

8.24 Returns

self
[object] The updated object.

class strlearn.ensembles.**NIE**(*base_estimator=None*, *n_estimators=5*, *param_a=1*, *param_b=1*)

Bases: *StreamingEnsemble*

Ditzler, Gregory, and Robi Polikar. "Incremental learning of concept drift from streaming imbalanced data." IEEE Transactions on Knowledge and Data Engineering 25.10 (2013): 2283-2301.

ensemble_support_matrix(*X*)

Ensemble support matrix.

partial_fit(*X*, *y*, *classes=None*)

Partial fitting

set_partial_fit_request(*, *classes: bool | None | str = '$UNCHANGED$'*) → *NIE*

Request metadata passed to the `partial_fit` method.

Note that this method is only relevant if `enable_metadata_routing=True` (see `sklearn.set_config()`). Please see User Guide on how the routing mechanism works.

The options for each parameter are:

- `True`: metadata is requested, and passed to `partial_fit` if provided. The request is ignored if metadata is not provided.
- `False`: metadata is not requested and the meta-estimator will not pass it to `partial_fit`.
- `None`: metadata is not requested, and the meta-estimator will raise an error if the user provides it.
- `str`: metadata should be passed to the meta-estimator with this given alias instead of the original name.

The default (`sklearn.utils.metadata_routing.UNCHANGED`) retains the existing request. This allows you to change the request for some parameters and not others.

Added in version 1.3.

Note: This method is only relevant if this estimator is used as a sub-estimator of a meta-estimator, e.g. used inside a `Pipeline`. Otherwise it has no effect.

8.25 Parameters

classes
: [str, True, False, or None, default=sklearn.utils.metadata_routing.UNCHANGED] Metadata routing for `classes` parameter in `partial_fit`.

8.26 Returns

self
: [object] The updated object.

set_score_request(*, *sample_weight: bool | None | str = '$UNCHANGED$'*) → *NIE*

Request metadata passed to the `score` method.

Note that this method is only relevant if `enable_metadata_routing=True` (see `sklearn.set_config()`). Please see User Guide on how the routing mechanism works.

The options for each parameter are:

- `True`: metadata is requested, and passed to `score` if provided. The request is ignored if metadata is not provided.
- `False`: metadata is not requested and the meta-estimator will not pass it to `score`.
- `None`: metadata is not requested, and the meta-estimator will raise an error if the user provides it.

- `str`: metadata should be passed to the meta-estimator with this given alias instead of the original name.

The default (`sklearn.utils.metadata_routing.UNCHANGED`) retains the existing request. This allows you to change the request for some parameters and not others.

Added in version 1.3.

Note: This method is only relevant if this estimator is used as a sub-estimator of a meta-estimator, e.g. used inside a `Pipeline`. Otherwise it has no effect.

8.27 Parameters

sample_weight
[str, True, False, or None, default=sklearn.utils.metadata_routing.UNCHANGED] Metadata routing for `sample_weight` parameter in `score`.

8.28 Returns

self
[object] The updated object.

class strlearn.ensembles.**OOB**(*base_estimator=None*, *n_estimators=5*, *time_decay_factor=0.9*)

Bases: *StreamingEnsemble*

Oversamping-Based Online Bagging.

partial_fit(*X*, *y*, *classes=None*)

Partial fitting

set_partial_fit_request(*, *classes: bool | None | str = '$UNCHANGED$'*) → *OOB*

Request metadata passed to the `partial_fit` method.

Note that this method is only relevant if `enable_metadata_routing=True` (see `sklearn.set_config()`). Please see User Guide on how the routing mechanism works.

The options for each parameter are:

- `True`: metadata is requested, and passed to `partial_fit` if provided. The request is ignored if metadata is not provided.
- `False`: metadata is not requested and the meta-estimator will not pass it to `partial_fit`.
- `None`: metadata is not requested, and the meta-estimator will raise an error if the user provides it.
- `str`: metadata should be passed to the meta-estimator with this given alias instead of the original name.

The default (`sklearn.utils.metadata_routing.UNCHANGED`) retains the existing request. This allows you to change the request for some parameters and not others.

Added in version 1.3.

Note: This method is only relevant if this estimator is used as a sub-estimator of a meta-estimator, e.g. used inside a `Pipeline`. Otherwise it has no effect.

8.29 Parameters

classes
: [str, True, False, or None, default=sklearn.utils.metadata_routing.UNCHANGED] Metadata routing for `classes` parameter in `partial_fit`.

8.30 Returns

self
: [object] The updated object.

`set_score_request`(*, *sample_weight: bool | None | str = '$UNCHANGED$'*) → *OOB*

Request metadata passed to the `score` method.

Note that this method is only relevant if `enable_metadata_routing=True` (see `sklearn.set_config()`). Please see User Guide on how the routing mechanism works.

The options for each parameter are:

- `True`: metadata is requested, and passed to `score` if provided. The request is ignored if metadata is not provided.
- `False`: metadata is not requested and the meta-estimator will not pass it to `score`.
- `None`: metadata is not requested, and the meta-estimator will raise an error if the user provides it.
- `str`: metadata should be passed to the meta-estimator with this given alias instead of the original name.

The default (`sklearn.utils.metadata_routing.UNCHANGED`) retains the existing request. This allows you to change the request for some parameters and not others.

Added in version 1.3.

Note: This method is only relevant if this estimator is used as a sub-estimator of a meta-estimator, e.g. used inside a `Pipeline`. Otherwise it has no effect.

8.31 Parameters

sample_weight
: [str, True, False, or None, default=sklearn.utils.metadata_routing.UNCHANGED] Metadata routing for `sample_weight` parameter in `score`.

8.32 Returns

self
[object] The updated object.

class strlearn.ensembles.**OUSE**(*base_estimator=None*, *n_estimators=10*, *n_chunks=10*)

Bases: ClassifierMixin, BaseEnsemble

Gao, Jing, et al. "Classifying Data Streams with Skewed Class Distributions and Concept Drifts." IEEE Internet Computing 12.6 (2008): 37-49.

ensemble_support_matrix(*X*)

Ensemble support matrix.

fit(*X*, *y*)

Fitting.

minority_majority_name(*y*)

Returns minority and majority data

Parameters
y (*array-like, shape (n_samples)*) – The target values.

Return type
tuple (object, object)

Returns
Tuple of minority and majority class names.

minority_majority_split(*X*, *y*, *minority_name*, *majority_name*)

Returns minority and majority data

Parameters

- **X** (*array-like, shape (n_samples, n_features)*) – The training input samples.
- **y** (*array-like, shape (n_samples)*) – The target values.

Return type
tuple (array-like, shape = [n_samples, n_features], array-like, shape = [n_samples, n_features])

Returns
Tuple of minority and majority class samples

partial_fit(*X*, *y*, *classes=None*)

Partial fitting.

predict(*X*)

Predict classes for X.

Parameters
X (*array-like, shape (n_samples, n_features)*) – The training input samples.

Return type
array-like, shape (n_samples,)

Returns
The predicted classes.

set_partial_fit_request(*, *classes: bool | None | str = '$UNCHANGED$'*) → *OUSE*

Request metadata passed to the `partial_fit` method.

Note that this method is only relevant if `enable_metadata_routing=True` (see `sklearn.set_config()`). Please see User Guide on how the routing mechanism works.

The options for each parameter are:

- `True`: metadata is requested, and passed to `partial_fit` if provided. The request is ignored if metadata is not provided.
- `False`: metadata is not requested and the meta-estimator will not pass it to `partial_fit`.
- `None`: metadata is not requested, and the meta-estimator will raise an error if the user provides it.
- `str`: metadata should be passed to the meta-estimator with this given alias instead of the original name.

The default (`sklearn.utils.metadata_routing.UNCHANGED`) retains the existing request. This allows you to change the request for some parameters and not others.

Added in version 1.3.

Note: This method is only relevant if this estimator is used as a sub-estimator of a meta-estimator, e.g. used inside a `Pipeline`. Otherwise it has no effect.

8.33 Parameters

classes
: [str, True, False, or None, default=sklearn.utils.metadata_routing.UNCHANGED] Metadata routing for `classes` parameter in `partial_fit`.

8.34 Returns

self
: [object] The updated object.

set_score_request(*, *sample_weight: bool | None | str = '$UNCHANGED$'*) → *OUSE*

Request metadata passed to the `score` method.

Note that this method is only relevant if `enable_metadata_routing=True` (see `sklearn.set_config()`). Please see User Guide on how the routing mechanism works.

The options for each parameter are:

- `True`: metadata is requested, and passed to `score` if provided. The request is ignored if metadata is not provided.
- `False`: metadata is not requested and the meta-estimator will not pass it to `score`.
- `None`: metadata is not requested, and the meta-estimator will raise an error if the user provides it.
- `str`: metadata should be passed to the meta-estimator with this given alias instead of the original name.

The default (`sklearn.utils.metadata_routing.UNCHANGED`) retains the existing request. This allows you to change the request for some parameters and not others.

Added in version 1.3.

Note: This method is only relevant if this estimator is used as a sub-estimator of a meta-estimator, e.g. used inside a `Pipeline`. Otherwise it has no effect.

8.35 Parameters

sample_weight
: [str, True, False, or None, default=sklearn.utils.metadata_routing.UNCHANGED] Metadata routing for `sample_weight` parameter in `score`.

8.36 Returns

self
: [object] The updated object.

class strlearn.ensembles.**OnlineBagging**(*base_estimator=None*, *n_estimators=10*)

Bases: *StreamingEnsemble*

Online Bagging.

partial_fit(*X*, *y*, *classes=None*)

Partial fitting

set_partial_fit_request(*, *classes: bool | None | str = '$UNCHANGED$'*) → *OnlineBagging*

Request metadata passed to the `partial_fit` method.

Note that this method is only relevant if `enable_metadata_routing=True` (see `sklearn.set_config()`). Please see User Guide on how the routing mechanism works.

The options for each parameter are:

- `True`: metadata is requested, and passed to `partial_fit` if provided. The request is ignored if metadata is not provided.
- `False`: metadata is not requested and the meta-estimator will not pass it to `partial_fit`.
- `None`: metadata is not requested, and the meta-estimator will raise an error if the user provides it.
- `str`: metadata should be passed to the meta-estimator with this given alias instead of the original name.

The default (`sklearn.utils.metadata_routing.UNCHANGED`) retains the existing request. This allows you to change the request for some parameters and not others.

Added in version 1.3.

Note: This method is only relevant if this estimator is used as a sub-estimator of a meta-estimator, e.g. used inside a `Pipeline`. Otherwise it has no effect.

8.37 Parameters

classes
: [str, True, False, or None, default=sklearn.utils.metadata_routing.UNCHANGED] Metadata routing for `classes` parameter in `partial_fit`.

8.38 Returns

self
: [object] The updated object.

`set_score_request`(*, *sample_weight: bool | None | str = '$UNCHANGED$'*) → *OnlineBagging*

Request metadata passed to the `score` method.

Note that this method is only relevant if `enable_metadata_routing=True` (see `sklearn.set_config()`). Please see User Guide on how the routing mechanism works.

The options for each parameter are:

- `True`: metadata is requested, and passed to `score` if provided. The request is ignored if metadata is not provided.
- `False`: metadata is not requested and the meta-estimator will not pass it to `score`.
- `None`: metadata is not requested, and the meta-estimator will raise an error if the user provides it.
- `str`: metadata should be passed to the meta-estimator with this given alias instead of the original name.

The default (`sklearn.utils.metadata_routing.UNCHANGED`) retains the existing request. This allows you to change the request for some parameters and not others.

Added in version 1.3.

Note: This method is only relevant if this estimator is used as a sub-estimator of a meta-estimator, e.g. used inside a `Pipeline`. Otherwise it has no effect.

8.39 Parameters

sample_weight
: [str, True, False, or None, default=sklearn.utils.metadata_routing.UNCHANGED] Metadata routing for `sample_weight` parameter in `score`.

8.40 Returns

self
[object] The updated object.

class strlearn.ensembles.**REA**(*base_estimator=None*, *n_estimators=10*, *post_balance_ratio=0.5*, *k_parameter=10*, *weighted=False*, *pruning=False*)

Bases: *StreamingEnsemble*

Recursive Ensemble Approach.

Sheng Chen, and Haibo He. "Towards incremental learning of nonstationary imbalanced data stream: a multiple selectively recursive approach." Evolving Systems 2.1 (2011): 35-50.

partial_fit(*X*, *y*, *classes=None*)

Partial fitting

set_partial_fit_request(*, *classes: bool | None | str = '$UNCHANGED$'*) → *REA*

Request metadata passed to the `partial_fit` method.

Note that this method is only relevant if `enable_metadata_routing=True` (see `sklearn.set_config()`). Please see User Guide on how the routing mechanism works.

The options for each parameter are:

- `True`: metadata is requested, and passed to `partial_fit` if provided. The request is ignored if metadata is not provided.
- `False`: metadata is not requested and the meta-estimator will not pass it to `partial_fit`.
- `None`: metadata is not requested, and the meta-estimator will raise an error if the user provides it.
- `str`: metadata should be passed to the meta-estimator with this given alias instead of the original name.

The default (`sklearn.utils.metadata_routing.UNCHANGED`) retains the existing request. This allows you to change the request for some parameters and not others.

Added in version 1.3.

Note: This method is only relevant if this estimator is used as a sub-estimator of a meta-estimator, e.g. used inside a `Pipeline`. Otherwise it has no effect.

8.41 Parameters

classes
[str, True, False, or None, default=sklearn.utils.metadata_routing.UNCHANGED] Metadata routing for `classes` parameter in `partial_fit`.

8.42 Returns

self
[object] The updated object.

set_score_request(*, *sample_weight: bool | None | str = '$UNCHANGED$'*) → *REA*

Request metadata passed to the `score` method.

Note that this method is only relevant if `enable_metadata_routing=True` (see `sklearn.set_config()`). Please see User Guide on how the routing mechanism works.

The options for each parameter are:

- `True`: metadata is requested, and passed to `score` if provided. The request is ignored if metadata is not provided.
- `False`: metadata is not requested and the meta-estimator will not pass it to `score`.
- `None`: metadata is not requested, and the meta-estimator will raise an error if the user provides it.
- `str`: metadata should be passed to the meta-estimator with this given alias instead of the original name.

The default (`sklearn.utils.metadata_routing.UNCHANGED`) retains the existing request. This allows you to change the request for some parameters and not others.

Added in version 1.3.

Note: This method is only relevant if this estimator is used as a sub-estimator of a meta-estimator, e.g. used inside a `Pipeline`. Otherwise it has no effect.

8.43 Parameters

sample_weight
[str, True, False, or None, default=sklearn.utils.metadata_routing.UNCHANGED] Metadata routing for `sample_weight` parameter in `score`.

8.44 Returns

self
[object] The updated object.

class strlearn.ensembles.**ROSE**(*base_estimator=None*, *n_estimators=10*, *n_candidates=1*, *subspace_mean=0.7*, *buffer_limit=1000*, *min_lambda=4*)

Bases: *StreamingEnsemble*

Robust Online Self-Adjusting Ensemble

ensemble_support_matrix(*X*)

Ensemble support matrix.

`partial_fit`(*X*, *y*, *classes=None*)

Partial fitting

`set_partial_fit_request`(**, classes: bool | None | str = '$UNCHANGED$'*) → *ROSE*

Request metadata passed to the `partial_fit` method.

Note that this method is only relevant if `enable_metadata_routing=True` (see `sklearn.set_config()`). Please see User Guide on how the routing mechanism works.

The options for each parameter are:

- `True`: metadata is requested, and passed to `partial_fit` if provided. The request is ignored if metadata is not provided.
- `False`: metadata is not requested and the meta-estimator will not pass it to `partial_fit`.
- `None`: metadata is not requested, and the meta-estimator will raise an error if the user provides it.
- `str`: metadata should be passed to the meta-estimator with this given alias instead of the original name.

The default (`sklearn.utils.metadata_routing.UNCHANGED`) retains the existing request. This allows you to change the request for some parameters and not others.

Added in version 1.3.

Note: This method is only relevant if this estimator is used as a sub-estimator of a meta-estimator, e.g. used inside a `Pipeline`. Otherwise it has no effect.

8.45 Parameters

classes
: [str, True, False, or None, default=sklearn.utils.metadata_routing.UNCHANGED] Metadata routing for `classes` parameter in `partial_fit`.

8.46 Returns

self
: [object] The updated object.

`set_score_request`(**, sample_weight: bool | None | str = '$UNCHANGED$'*) → *ROSE*

Request metadata passed to the `score` method.

Note that this method is only relevant if `enable_metadata_routing=True` (see `sklearn.set_config()`). Please see User Guide on how the routing mechanism works.

The options for each parameter are:

- `True`: metadata is requested, and passed to `score` if provided. The request is ignored if metadata is not provided.
- `False`: metadata is not requested and the meta-estimator will not pass it to `score`.
- `None`: metadata is not requested, and the meta-estimator will raise an error if the user provides it.

- `str`: metadata should be passed to the meta-estimator with this given alias instead of the original name.

The default (`sklearn.utils.metadata_routing.UNCHANGED`) retains the existing request. This allows you to change the request for some parameters and not others.

Added in version 1.3.

Note: This method is only relevant if this estimator is used as a sub-estimator of a meta-estimator, e.g. used inside a `Pipeline`. Otherwise it has no effect.

8.47 Parameters

sample_weight
[str, True, False, or None, default=sklearn.utils.metadata_routing.UNCHANGED] Metadata routing for `sample_weight` parameter in `score`.

8.48 Returns

self
[object] The updated object.

class strlearn.ensembles.**SEA**(*base_estimator=None*, *n_estimators=10*, *metric=<function accuracy_score>*)

Bases: *StreamingEnsemble*

Streaming Ensemble Algorithm.

Ensemble classifier composed of estimators trained on the fixed number of previously seen data chunks, prunning the worst one in the pool.

Parameters

- **n_estimators** (*integer, optional (default=10)*) – The maximum number of estimators trained using consecutive data chunks and maintained in the ensemble.
- **metric** (*function, optional (default=accuracy_score)*) – The metric used to prune the worst classifier in the pool.

Variables

- **ensemble** (*list of classifiers*) – The collection of fitted sub-estimators.
- **classes** (*array-like, shape (n_classes,)*) – The class labels.

Example

```
>>> import strlearn as sl
>>> stream = sl.streams.StreamGenerator()
>>> clf = sl.ensembles.SEA()
>>> evaluator = sl.evaluators.TestThenTrainEvaluator()
>>> evaluator.process(clf, stream)
>>> print(evaluator.scores_)
...
```

(continues on next page)

(continued from previous page)

```
[[0.92       0.91879699 0.91848191 0.91879699 0.92523364]
[0.945      0.94648779 0.94624912 0.94648779 0.94240838]
[0.925      0.92364329 0.92360881 0.92364329 0.91017964]
...
[0.925      0.92427885 0.924103   0.92427885 0.92890995]
[0.89       0.89016179 0.89015879 0.89016179 0.88297872]
[0.935      0.93569212 0.93540766 0.93569212 0.93467337]]
```

partial_fit(*X*, *y*, *classes=None*)

Partial fitting

set_partial_fit_request(*, *classes: bool | None | str = '$UNCHANGED$'*) → *SEA*

Request metadata passed to the `partial_fit` method.

Note that this method is only relevant if `enable_metadata_routing=True` (see `sklearn.set_config()`). Please see User Guide on how the routing mechanism works.

The options for each parameter are:

- `True`: metadata is requested, and passed to `partial_fit` if provided. The request is ignored if metadata is not provided.
- `False`: metadata is not requested and the meta-estimator will not pass it to `partial_fit`.
- `None`: metadata is not requested, and the meta-estimator will raise an error if the user provides it.
- `str`: metadata should be passed to the meta-estimator with this given alias instead of the original name.

The default (`sklearn.utils.metadata_routing.UNCHANGED`) retains the existing request. This allows you to change the request for some parameters and not others.

Added in version 1.3.

Note: This method is only relevant if this estimator is used as a sub-estimator of a meta-estimator, e.g. used inside a `Pipeline`. Otherwise it has no effect.

8.49 Parameters

classes

[str, True, False, or None, default=sklearn.utils.metadata_routing.UNCHANGED] Metadata routing for `classes` parameter in `partial_fit`.

8.50 Returns

self
[object] The updated object.

`set_score_request`(*, *sample_weight: bool | None | str = '$UNCHANGED$'*) → *SEA*

Request metadata passed to the `score` method.

Note that this method is only relevant if `enable_metadata_routing=True` (see `sklearn.set_config()`). Please see User Guide on how the routing mechanism works.

The options for each parameter are:

- `True`: metadata is requested, and passed to `score` if provided. The request is ignored if metadata is not provided.
- `False`: metadata is not requested and the meta-estimator will not pass it to `score`.
- `None`: metadata is not requested, and the meta-estimator will raise an error if the user provides it.
- `str`: metadata should be passed to the meta-estimator with this given alias instead of the original name.

The default (`sklearn.utils.metadata_routing.UNCHANGED`) retains the existing request. This allows you to change the request for some parameters and not others.

Added in version 1.3.

Note: This method is only relevant if this estimator is used as a sub-estimator of a meta-estimator, e.g. used inside a `Pipeline`. Otherwise it has no effect.

8.51 Parameters

sample_weight
[str, True, False, or None, default=sklearn.utils.metadata_routing.UNCHANGED] Metadata routing for `sample_weight` parameter in `score`.

8.52 Returns

self
[object] The updated object.

`class strlearn.ensembles.`**`StreamingEnsemble`**(*base_estimator*, *n_estimators*, *weighted=False*)

Bases: `ClassifierMixin`, `BaseEstimator`

Abstract, base ensemble streaming class

`ensemble_support_matrix`(*X*)

Ensemble support matrix.

`fit`(*X*, *y*)

Fitting.

minority_majority_name(*y*)

Returns minority and majority data

Parameters
y (*array-like, shape (n_samples)*) – The target values.

Return type
tuple (object, object)

Returns
Tuple of minority and majority class names.

minority_majority_split(*X*, *y*, *minority_name*, *majority_name*)

Returns minority and majority data

Parameters

- **X** (*array-like, shape (n_samples, n_features)*) – The training input samples.
- **y** (*array-like, shape (n_samples)*) – The target values.

Return type
tuple (array-like, shape = [n_samples, n_features], array-like, shape = [n_samples, n_features])

Returns
Tuple of minority and majority class samples

msei(*clf*, *X*, *y*)

MSEi score from original AWE algorithm.

mser(*y*)

MSEr score from original AWE algorithm.

partial_fit(*X*, *y*, *classes=None*)

Partial fitting

predict(*X*)

Predict classes for X.

Parameters
X (*array-like, shape (n_samples, n_features)*) – The training input samples.

Return type
array-like, shape (n_samples,)

Returns
The predicted classes.

predict_proba(*X*)

Predict proba.

prior_proba(*y*)

Calculate prior probability for given labels

set_partial_fit_request(**, classes: bool | None | str = '$UNCHANGED$'*) → *StreamingEnsemble*

Request metadata passed to the `partial_fit` method.

Note that this method is only relevant if `enable_metadata_routing=True` (see `sklearn.set_config()`). Please see User Guide on how the routing mechanism works.

The options for each parameter are:

- `True`: metadata is requested, and passed to `partial_fit` if provided. The request is ignored if metadata is not provided.
- `False`: metadata is not requested and the meta-estimator will not pass it to `partial_fit`.
- `None`: metadata is not requested, and the meta-estimator will raise an error if the user provides it.
- `str`: metadata should be passed to the meta-estimator with this given alias instead of the original name.

The default (`sklearn.utils.metadata_routing.UNCHANGED`) retains the existing request. This allows you to change the request for some parameters and not others.

Added in version 1.3.

Note: This method is only relevant if this estimator is used as a sub-estimator of a meta-estimator, e.g. used inside a `Pipeline`. Otherwise it has no effect.

8.53 Parameters

classes
: [str, True, False, or None, default=sklearn.utils.metadata_routing.UNCHANGED] Metadata routing for `classes` parameter in `partial_fit`.

8.54 Returns

self
: [object] The updated object.

`set_score_request`(*, *sample_weight: bool | None | str = '$UNCHANGED$'*) → *StreamingEnsemble*

Request metadata passed to the `score` method.

Note that this method is only relevant if `enable_metadata_routing=True` (see `sklearn.set_config()`). Please see User Guide on how the routing mechanism works.

The options for each parameter are:

- `True`: metadata is requested, and passed to `score` if provided. The request is ignored if metadata is not provided.
- `False`: metadata is not requested and the meta-estimator will not pass it to `score`.
- `None`: metadata is not requested, and the meta-estimator will raise an error if the user provides it.
- `str`: metadata should be passed to the meta-estimator with this given alias instead of the original name.

The default (`sklearn.utils.metadata_routing.UNCHANGED`) retains the existing request. This allows you to change the request for some parameters and not others.

Added in version 1.3.

Note: This method is only relevant if this estimator is used as a sub-estimator of a meta-estimator, e.g. used inside a `Pipeline`. Otherwise it has no effect.

8.55 Parameters

sample_weight
[str, True, False, or None, default=sklearn.utils.metadata_routing.UNCHANGED] Metadata routing for `sample_weight` parameter in `score`.

8.56 Returns

self
[object] The updated object.

class strlearn.ensembles.**UOB**(*base_estimator=None*, *n_estimators=5*, *time_decay_factor=0.9*)

Bases: *StreamingEnsemble*

Undersampling-Based Online Bagging.

partial_fit(*X*, *y*, *classes=None*)

Partial fitting

set_partial_fit_request(*, *classes: bool | None | str = '$UNCHANGED$'*) → *UOB*

Request metadata passed to the `partial_fit` method.

Note that this method is only relevant if `enable_metadata_routing=True` (see `sklearn.set_config()`). Please see User Guide on how the routing mechanism works.

The options for each parameter are:

- `True`: metadata is requested, and passed to `partial_fit` if provided. The request is ignored if metadata is not provided.
- `False`: metadata is not requested and the meta-estimator will not pass it to `partial_fit`.
- `None`: metadata is not requested, and the meta-estimator will raise an error if the user provides it.
- `str`: metadata should be passed to the meta-estimator with this given alias instead of the original name.

The default (`sklearn.utils.metadata_routing.UNCHANGED`) retains the existing request. This allows you to change the request for some parameters and not others.

Added in version 1.3.

Note: This method is only relevant if this estimator is used as a sub-estimator of a meta-estimator, e.g. used inside a `Pipeline`. Otherwise it has no effect.

8.57 Parameters

classes
[str, True, False, or None, default=sklearn.utils.metadata_routing.UNCHANGED] Metadata routing for `classes` parameter in `partial_fit`.

8.58 Returns

self
[object] The updated object.

`set_score_request`(*, *sample_weight: bool | None | str = '$UNCHANGED$'*) → *UOB*

Request metadata passed to the `score` method.

Note that this method is only relevant if `enable_metadata_routing=True` (see `sklearn.set_config()`). Please see User Guide on how the routing mechanism works.

The options for each parameter are:

- `True`: metadata is requested, and passed to `score` if provided. The request is ignored if metadata is not provided.
- `False`: metadata is not requested and the meta-estimator will not pass it to `score`.
- `None`: metadata is not requested, and the meta-estimator will raise an error if the user provides it.
- `str`: metadata should be passed to the meta-estimator with this given alias instead of the original name.

The default (`sklearn.utils.metadata_routing.UNCHANGED`) retains the existing request. This allows you to change the request for some parameters and not others.

Added in version 1.3.

Note: This method is only relevant if this estimator is used as a sub-estimator of a meta-estimator, e.g. used inside a `Pipeline`. Otherwise it has no effect.

8.59 Parameters

sample_weight
[str, True, False, or None, default=sklearn.utils.metadata_routing.UNCHANGED] Metadata routing for `sample_weight` parameter in `score`.

8.60 Returns

self
[object] The updated object.

class strlearn.ensembles.**WAE**(*base_estimator=None*, *n_estimators=10*, *theta=0.1*, *post_pruning=False*, *pruning_criterion='accuracy'*, *weight_calculation_method='kuncheva'*, *aging_method='weights_proportional'*, *rejuvenation_power=0.0*)

Bases: *StreamingEnsemble*

Weighted Aging Ensemble.

The method was inspired by Accuracy Weighted Ensemble (AWE) algorithm to which it introduces two main modifications: (I) classifier weights depend on the individual classifier accuracies and time they have been spending in the ensemble, (II) individual classifier are chosen on the basis on the non-pairwise diversity measure.

Parameters

- **base_estimator** (*ClassifierMixin class object*) – Classification algorithm used as a base estimator.
- **n_estimators** (*integer, optional (default=10)*) – The maximum number of estimators trained using consecutive data chunks and maintained in the ensemble.
- **theta** (*float, optional (default=0.1)*) – Threshold for weight calculation method and aging procedure control.
- **post_pruning** (*boolean, optional (default=False)*) – Whether the pruning is conducted before or after adding the classifier.
- **pruning_criterion** (*string, optional (default='accuracy')*) – Selection of pruning criterion.
- **weight_calculation_method** (*string, optional (default='kuncheva')*) – same_for_each, proportional_to_accuracy, kuncheva, pta_related_to_whole, bell_curve,
- **aging_method** (*string, optional (default='weights_proportional')*) – weights_proportional, constant, gaussian.
- **rejuvenation_power** (*float, optional (default=0.0)*) – Rejuvenation dynamics control of classifiers with high prediction accuracy.

Variables

- **ensemble** (*list of classifiers*) – The collection of fitted sub-estimators.
- **classes** (*array-like, shape (n_classes,)*) – The class labels.
- **weights** (*array-like, shape (n_estimators,)*) – Classifier weights.

Examples

```
>>> import strlearn as sl
>>> from sklearn.naive_bayes import GaussianNB
>>> stream = sl.streams.StreamGenerator()
>>> clf = sl.ensembles.WAE(GaussianNB())
>>> ttt = sl.evaluators.TestThenTrain(
>>> metrics=(sl.metrics.balanced_accuracy_score))
>>> ttt.process(stream, clf)
>>> print(ttt.scores)
```

(continues on next page)

(continued from previous page)

```
[[[0.91386218]
  [0.93032581]
  [0.90907219]
  [0.90544872]
  [0.90466186]
  [0.91956783]
  [0.90776942]
  [0.92685422]
  [0.92895186]
  ...
```

partial_fit(*X*, *y*, *classes=None*)

Partial fitting

set_partial_fit_request(***, *classes: bool | None | str = '$UNCHANGED$'*) → *WAE*

Request metadata passed to the `partial_fit` method.

Note that this method is only relevant if `enable_metadata_routing=True` (see `sklearn.set_config()`). Please see User Guide on how the routing mechanism works.

The options for each parameter are:

- `True`: metadata is requested, and passed to `partial_fit` if provided. The request is ignored if metadata is not provided.
- `False`: metadata is not requested and the meta-estimator will not pass it to `partial_fit`.
- `None`: metadata is not requested, and the meta-estimator will raise an error if the user provides it.
- `str`: metadata should be passed to the meta-estimator with this given alias instead of the original name.

The default (`sklearn.utils.metadata_routing.UNCHANGED`) retains the existing request. This allows you to change the request for some parameters and not others.

Added in version 1.3.

Note: This method is only relevant if this estimator is used as a sub-estimator of a meta-estimator, e.g. used inside a `Pipeline`. Otherwise it has no effect.

8.61 Parameters

classes

[str, True, False, or None, default=sklearn.utils.metadata_routing.UNCHANGED] Metadata routing for `classes` parameter in `partial_fit`.

8.62 Returns

self
[object] The updated object.

`set_score_request`(*, *sample_weight: bool | None | str = '$UNCHANGED$'*) → *WAE*

Request metadata passed to the `score` method.

Note that this method is only relevant if `enable_metadata_routing=True` (see `sklearn.set_config()`). Please see User Guide on how the routing mechanism works.

The options for each parameter are:

- `True`: metadata is requested, and passed to `score` if provided. The request is ignored if metadata is not provided.
- `False`: metadata is not requested and the meta-estimator will not pass it to `score`.
- `None`: metadata is not requested, and the meta-estimator will raise an error if the user provides it.
- `str`: metadata should be passed to the meta-estimator with this given alias instead of the original name.

The default (`sklearn.utils.metadata_routing.UNCHANGED`) retains the existing request. This allows you to change the request for some parameters and not others.

Added in version 1.3.

Note: This method is only relevant if this estimator is used as a sub-estimator of a meta-estimator, e.g. used inside a `Pipeline`. Otherwise it has no effect.

8.63 Parameters

sample_weight
[str, True, False, or None, default=sklearn.utils.metadata_routing.UNCHANGED] Metadata routing for `sample_weight` parameter in `score`.

8.64 Returns

self
[object] The updated object.

CHAPTER

NINE

CLASSIFIERS MODULE

ASC([base_clf])	Accumulated samples classifier.
SampleWeightedMetaEstimator([base_classifier])	Sample Weighted Meta Estimator.

class strlearn.classifiers.**ASC**(*base_clf=None*)

Bases: BaseEnsemble, ClassifierMixin

Accumulated samples classifier.

Classifier fitted on accumulated samples from all data chunks.

Variables

classes (*array-like, shape (n_classes,)*) – The class labels.

Example

```
>>> import strlearn as sl
>>> stream = sl.streams.StreamGenerator()
>>> clf = sl.classifiers.AccumulatedSamplesClassifier()
>>> evaluator = sl.evaluators.TestThenTrainEvaluator()
>>> evaluator.process(clf, stream)
>>> print(evaluator.scores_)
...
[[0.92       0.91879699 0.91848191 0.91879699 0.92523364]
[0.945      0.94648779 0.94624912 0.94648779 0.94240838]
[0.92       0.91936979 0.91936231 0.91936979 0.9047619 ]
...
[0.92       0.91907051 0.91877671 0.91907051 0.9245283 ]
[0.885      0.8854889  0.88546135 0.8854889  0.87830688]
[0.935      0.93569212 0.93540766 0.93569212 0.93467337]]
```

fit(*X*, *y*)

Fitting.

partial_fit(*X*, *y*, *classes=None*)

Partial fitting.

set_partial_fit_request(*, *classes: bool | None | str = '$UNCHANGED$'*) → *ASC*

Request metadata passed to the partial_fit method.

Note that this method is only relevant if enable_metadata_routing=True (see sklearn.set_config()). Please see User Guide on how the routing mechanism works.

The options for each parameter are:

- `True`: metadata is requested, and passed to `partial_fit` if provided. The request is ignored if metadata is not provided.
- `False`: metadata is not requested and the meta-estimator will not pass it to `partial_fit`.
- `None`: metadata is not requested, and the meta-estimator will raise an error if the user provides it.
- `str`: metadata should be passed to the meta-estimator with this given alias instead of the original name.

The default (`sklearn.utils.metadata_routing.UNCHANGED`) retains the existing request. This allows you to change the request for some parameters and not others.

Added in version 1.3.

Note: This method is only relevant if this estimator is used as a sub-estimator of a meta-estimator, e.g. used inside a `Pipeline`. Otherwise it has no effect.

9.1 Parameters

classes
: [str, True, False, or None, default=sklearn.utils.metadata_routing.UNCHANGED] Metadata routing for `classes` parameter in `partial_fit`.

9.2 Returns

self
: [object] The updated object.

`set_score_request`(*, *sample_weight: bool | None | str = '$UNCHANGED$'*) → *ASC*

Request metadata passed to the `score` method.

Note that this method is only relevant if `enable_metadata_routing=True` (see `sklearn.set_config()`). Please see User Guide on how the routing mechanism works.

The options for each parameter are:

- `True`: metadata is requested, and passed to `score` if provided. The request is ignored if metadata is not provided.
- `False`: metadata is not requested and the meta-estimator will not pass it to `score`.
- `None`: metadata is not requested, and the meta-estimator will raise an error if the user provides it.
- `str`: metadata should be passed to the meta-estimator with this given alias instead of the original name.

The default (`sklearn.utils.metadata_routing.UNCHANGED`) retains the existing request. This allows you to change the request for some parameters and not others.

Added in version 1.3.

Note: This method is only relevant if this estimator is used as a sub-estimator of a meta-estimator, e.g. used inside a `Pipeline`. Otherwise it has no effect.

9.3 Parameters

sample_weight
[str, True, False, or None, default=sklearn.utils.metadata_routing.UNCHANGED] Metadata routing for `sample_weight` parameter in `score`.

9.4 Returns

self
[object] The updated object.

`class strlearn.classifiers.SampleWeightedMetaEstimator(`*base_classifier=GaussianNB()*`)`

Bases: `BaseEstimator`, `ClassifierMixin`

Sample Weighted Meta Estimator.

`set_partial_fit_request(`*, classes: bool | None | str = '$UNCHANGED$', sample_weight: bool | None | str = '$UNCHANGED$'*`)` → *SampleWeightedMetaEstimator*

Request metadata passed to the `partial_fit` method.

Note that this method is only relevant if `enable_metadata_routing=True` (see `sklearn.set_config()`). Please see User Guide on how the routing mechanism works.

The options for each parameter are:

- `True`: metadata is requested, and passed to `partial_fit` if provided. The request is ignored if metadata is not provided.
- `False`: metadata is not requested and the meta-estimator will not pass it to `partial_fit`.
- `None`: metadata is not requested, and the meta-estimator will raise an error if the user provides it.
- `str`: metadata should be passed to the meta-estimator with this given alias instead of the original name.

The default (`sklearn.utils.metadata_routing.UNCHANGED`) retains the existing request. This allows you to change the request for some parameters and not others.

Added in version 1.3.

Note: This method is only relevant if this estimator is used as a sub-estimator of a meta-estimator, e.g. used inside a `Pipeline`. Otherwise it has no effect.

9.5 Parameters

classes
[str, True, False, or None, default=sklearn.utils.metadata_routing.UNCHANGED] Metadata routing for `classes` parameter in `partial_fit`.

sample_weight
[str, True, False, or None, default=sklearn.utils.metadata_routing.UNCHANGED] Metadata routing for `sample_weight` parameter in `partial_fit`.

9.6 Returns

self
[object] The updated object.

`set_score_request`(*, *sample_weight: bool | None | str = '$UNCHANGED$'*) → *SampleWeightedMetaEstimator*

Request metadata passed to the `score` method.

Note that this method is only relevant if `enable_metadata_routing=True` (see `sklearn.set_config()`). Please see User Guide on how the routing mechanism works.

The options for each parameter are:

- `True`: metadata is requested, and passed to `score` if provided. The request is ignored if metadata is not provided.
- `False`: metadata is not requested and the meta-estimator will not pass it to `score`.
- `None`: metadata is not requested, and the meta-estimator will raise an error if the user provides it.
- `str`: metadata should be passed to the meta-estimator with this given alias instead of the original name.

The default (`sklearn.utils.metadata_routing.UNCHANGED`) retains the existing request. This allows you to change the request for some parameters and not others.

Added in version 1.3.

Note: This method is only relevant if this estimator is used as a sub-estimator of a meta-estimator, e.g. used inside a `Pipeline`. Otherwise it has no effect.

9.7 Parameters

sample_weight
[str, True, False, or None, default=sklearn.utils.metadata_routing.UNCHANGED] Metadata routing for `sample_weight` parameter in `score`.

9.8 Returns

self
[object] The updated object.

`class` strlearn.classifiers.**Skipper**(*base_clf*, *n_skips=5*)

Bases: `BaseEstimator`, `ClassifierMixin`

Skipper.

fit(*X*, *y*)

Fitting.

set_partial_fit_request(*, *classes: bool | None | str = '$UNCHANGED$'*) → *Skipper*

Request metadata passed to the `partial_fit` method.

Note that this method is only relevant if `enable_metadata_routing=True` (see `sklearn.set_config()`). Please see User Guide on how the routing mechanism works.

The options for each parameter are:

- `True`: metadata is requested, and passed to `partial_fit` if provided. The request is ignored if metadata is not provided.
- `False`: metadata is not requested and the meta-estimator will not pass it to `partial_fit`.
- `None`: metadata is not requested, and the meta-estimator will raise an error if the user provides it.
- `str`: metadata should be passed to the meta-estimator with this given alias instead of the original name.

The default (`sklearn.utils.metadata_routing.UNCHANGED`) retains the existing request. This allows you to change the request for some parameters and not others.

Added in version 1.3.

Note: This method is only relevant if this estimator is used as a sub-estimator of a meta-estimator, e.g. used inside a `Pipeline`. Otherwise it has no effect.

9.9 Parameters

classes
: [str, True, False, or None, default=sklearn.utils.metadata_routing.UNCHANGED] Metadata routing for `classes` parameter in `partial_fit`.

9.10 Returns

self
: [object] The updated object.

set_score_request(*, *sample_weight: bool | None | str = '$UNCHANGED$'*) → *Skipper*

Request metadata passed to the `score` method.

Note that this method is only relevant if `enable_metadata_routing=True` (see `sklearn.set_config()`). Please see User Guide on how the routing mechanism works.

The options for each parameter are:

- `True`: metadata is requested, and passed to `score` if provided. The request is ignored if metadata is not provided.
- `False`: metadata is not requested and the meta-estimator will not pass it to `score`.
- `None`: metadata is not requested, and the meta-estimator will raise an error if the user provides it.
- `str`: metadata should be passed to the meta-estimator with this given alias instead of the original name.

The default (`sklearn.utils.metadata_routing.UNCHANGED`) retains the existing request. This allows you to change the request for some parameters and not others.

Added in version 1.3.

Note: This method is only relevant if this estimator is used as a sub-estimator of a meta-estimator, e.g. used inside a `Pipeline`. Otherwise it has no effect.

9.11 Parameters

sample_weight
: [str, True, False, or None, default=sklearn.utils.metadata_routing.UNCHANGED] Metadata routing for `sample_weight` parameter in `score`.

9.12 Returns

self
: [object] The updated object.

CHAPTER

TEN

UTILS MODULE

scores_to_cummean(scores)	Convert evaluator scores to accumulative mean.

strlearn.utils.**scores_to_cummean**(*scores*)

Convert evaluator scores to accumulative mean.

It's the best way to make reader capable to understand anything from your results.

Parameters

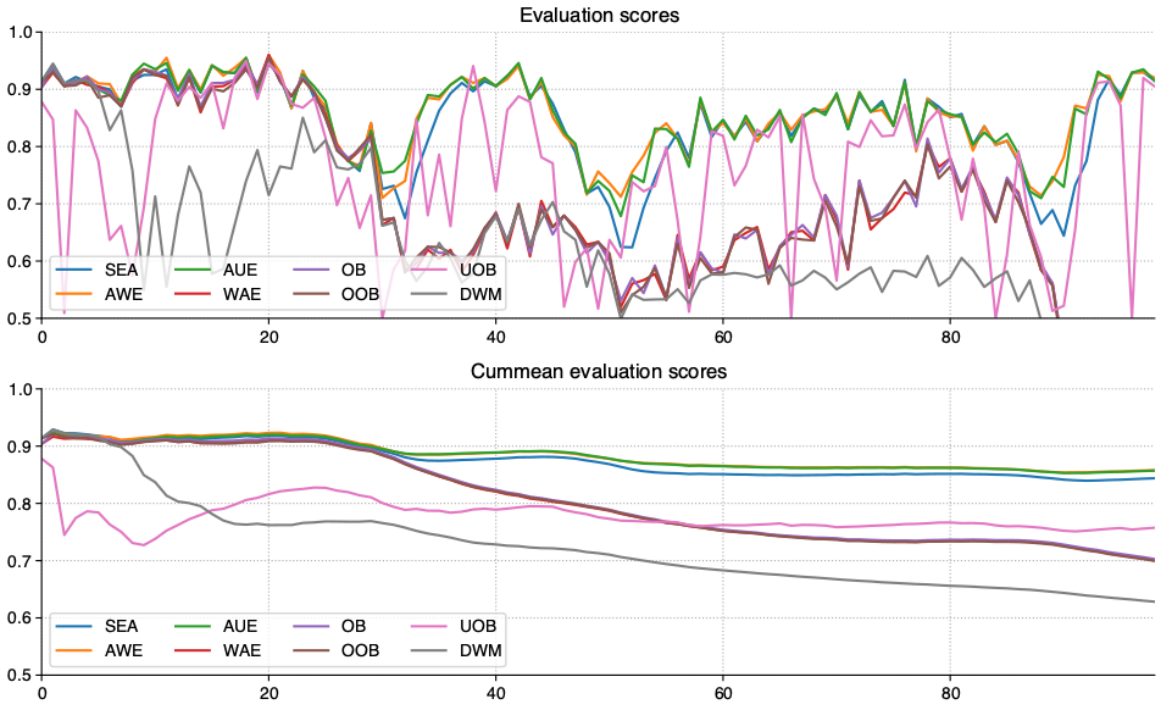
scores (*array-like, shape (n_estimators, n_chunks, n_metrics)*) – Evaluation scores.

Return type

array-like, shape (n_estimators, n_chunks, n_metrics)

Returns

Evaluation scores in format possible to read for human being.



CHAPTER

ELEVEN

METRICS MODULE

binary_confusion_matrix(y_true, y_pred)	Calculates the binary confusion matrics.
specificity(y_true, y_pred)	Calculates the specificity.
recall(y_true, y_pred)	Calculates the recall.
precision(y_true, y_pred)	Calculates the precision.
fbeta_score(y_true, y_pred, beta)	Calculates the F-beta score.
f1_score(y_true, y_pred)	Calculates the f1_score.
balanced_accuracy_score(y_true, y_pred)	Calculates the balanced accuracy score.
geometric_mean_score_1(y_true, y_pred)	Calculates the geometric mean score.
geometric_mean_score_2(y_true, y_pred)	Calculates the geometric mean score.

strlearn.metrics.**balanced_accuracy_score**(*y_true*, *y_pred*)

Calculates the balanced accuracy score.

The balanced accuracy for the multiclass problems is defined as the average of recall obtained on each class. For binary problems it is denoted by the average of recall and specificity (also called true negative rate).

$$BAC = \frac{Recall + Specificity}{2}$$

Parameters

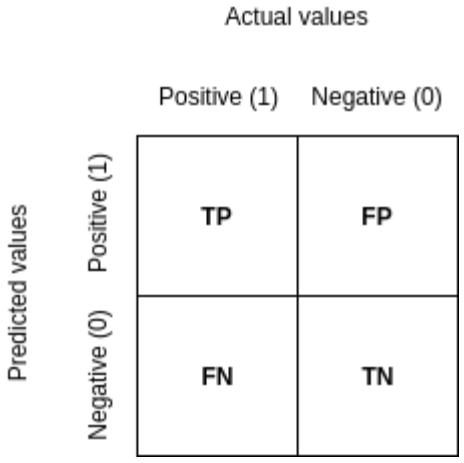
- **y_true** (*array-like, shape (n_samples)*) – True labels.
- **y_pred** (*array-like, shape (n_samples)*) – Predicted labels.

Return type
float

Returns
Balanced accuracy score.

strlearn.metrics.**binary_confusion_matrix**(*y_true*, *y_pred*)

Calculates the binary confusion matrics.



Parameters

- **y_true** (*array-like, shape (n_samples)*) – True labels.
- **y_pred** (*array-like, shape (n_samples)*) – Predicted labels.

Return type
tuple, (TN, FP, FN, TP)

Returns
Elements of binary confusion matrix.

strlearn.metrics.**f1_score**(*y_true*, *y_pred*)

Calculates the f1_score.

The F1 score can be interpreted as a F-beta score, where eta parameter equals 1. It is a harmonic mean of precision and recall. The formula for the F1 score is

$$F_1 = 2 * \frac{Precision * Recall}{Precision + Recall}$$

Parameters

- **y_true** (*array-like, shape (n_samples)*) – True labels.
- **y_pred** (*array-like, shape (n_samples)*) – Predicted labels.

Return type
float

Returns
F1 score.

strlearn.metrics.**fbeta_score**(*y_true*, *y_pred*, *beta*)

Calculates the F-beta score.

The F-beta score can be interpreted as a weighted harmonic mean of precision and recall taking both metrics into account and punishing extreme values. The `beta` parameter determines the recall's weight. `beta` < 1 gives more weight to precision, while `beta` > 1 prefers recall. The formula for the F-beta score is

$$F_\beta = (1 + \beta^2) * \frac{Precision * Recall}{(\beta^2 * Precision) + Recall}$$

Parameters

- **y_true** (*array-like, shape (n_samples)*) – True labels.

- **y_pred** (*array-like, shape (n_samples)*) – Predicted labels.
- **beta** (*float*) – Beta parameter

Return type
float

Returns
F-beta score.

strlearn.metrics.**geometric_mean_score_1**(*y_true*, *y_pred*)

Calculates the geometric mean score.

The geometric mean (G-mean) tries to maximize the accuracy on each of the classes while keeping these accuracies balanced. For N-class problems it is a N root of the product of class-wise recall. For binary classification G-mean is denoted as the squared root of the product of the recall and specificity.

$$Gmean1 = \sqrt{Recall * Specificity}$$

Parameters

- **y_true** (*array-like, shape (n_samples)*) – True labels.
- **y_pred** (*array-like, shape (n_samples)*) – Predicted labels.

Return type
float

Returns
Geometric mean score.

strlearn.metrics.**geometric_mean_score_2**(*y_true*, *y_pred*)

Calculates the geometric mean score.

The alternative definition of G-mean measure. For binary classification G-mean is denoted as the squared root of the product of the recall and precision.

$$Gmean2 = \sqrt{Recall * Precision}$$

Parameters

- **y_true** (*array-like, shape (n_samples)*) – True labels.
- **y_pred** (*array-like, shape (n_samples)*) – Predicted labels.

Return type
float

Returns
Geometric mean score.

strlearn.metrics.**precision**(*y_true*, *y_pred*)

Calculates the precision.

Precision (also called positive predictive value) expresses the probability of correct detection of positive samples and is denoted as

$$Precision = \frac{tp}{tp + fp}$$

Parameters

- **y_true** (*array-like, shape (n_samples)*) – True labels.

- **y_pred** (*array-like, shape (n_samples)*) – Predicted labels.

Return type
float

Returns
Precision score.

strlearn.metrics.**recall**(*y_true*, *y_pred*)

Calculates the recall.

Recall (also known as sensitivity or true positive rate) represents the classifier's ability to find all the positive data samples in the dataset (e.g. the minority class instances) and is denoted as

$$Recall = \frac{tp}{tp + fn}$$

Parameters

- **y_true** (*array-like, shape (n_samples)*) – True labels.
- **y_pred** (*array-like, shape (n_samples)*) – Predicted labels.

Return type
float

Returns
Recall score.

strlearn.metrics.**specificity**(*y_true*, *y_pred*)

Calculates the specificity.

$$Specificity = \frac{tn}{tn + fp}$$

Parameters

- **y_true** (*array-like, shape (n_samples)*) – True labels.
- **y_pred** (*array-like, shape (n_samples)*) – Predicted labels.

Return type
float

Returns
Specificity score.

CHAPTER

TWELVE

ABOUT US

The `stream-learn` package was created for the needs of the Department of Systems and Computer Networks, *Wrocław University of Science and Technology*, as part of research projects regarding the processing of imbalanced data streams and its code is used for experimental evaluation since 2017.

The authors and maintainers of its current version are the employees of the unit, namely P. Ksieniewicz and P. Zyblewski.

CHAPTER

THIRTEEN

CITATION POLICY

If you use `stream-learn` in a scientific publication, we would appreciate citation to the following paper:

```
@article{Ksieniewicz2022,
doi = {10.1016/j.neucom.2021.10.120},
url = {https://doi.org/10.1016/j.neucom.2021.10.120},
year = {2022},
month = jan,
publisher = {Elsevier {BV}},
author = {P. Ksieniewicz and P. Zyblewski},
title = {stream-learn {\textemdash} open-source Python library for difficult data stream␣
↪batch analysis},
journal = {Neurocomputing}
}
```

The `stream-learn` module is a set of tools necessary for processing data streams using `scikit-learn` estimators. The batch processing approach is used here, where the dataset is passed to the classifier in smaller, consecutive subsets called *chunks*. The module consists of five sub-modules:

- streams - containing a data stream generator that allows obtaining both stationary and dynamic distributions in accordance with various types of concept drift (also in the field of a priori probability, i.e. dynamically unbalanced data) and a parser of the standard ARFF file format.
- evaluators - containing classes for running experiments on stream data in accordance with the Test-Then-Train and Prequential methodology.
- classifiers - containing sample stream classifiers,
- ensembles - containing standard team models of stream data classification,
- metrics - containing typical classification quality metrics in data streams.

You can read more about each module in the User Guide.

CHAPTER

FOURTEEN

GETTING STARTED

A brief description of the installation process and basic usage of the module in a simple experiment.

CHAPTER

FIFTEEN

API DOCUMENTATION

Precise API description of all the classes and functions implemented in the module.

CHAPTER

SIXTEEN

EXAMPLES

A set of examples illustrating the use of all module elements.

See the README for more information.

PYTHON MODULE INDEX

S

strlearn.classifiers, 63
strlearn.ensembles, 31
strlearn.evaluators, 27
strlearn.metrics, 71
strlearn.streams, 21
strlearn.utils, 69

INDEX

A

ARFFParser (*class in strlearn.streams*), 21
ASC (*class in strlearn.classifiers*), 63
AUE (*class in strlearn.ensembles*), 31
AWE (*class in strlearn.ensembles*), 33

B

balanced_accuracy_score() (*in module strlearn.metrics*), 71
binary_confusion_matrix() (*in module strlearn.metrics*), 71

C

CDS (*class in strlearn.ensembles*), 35
CSVParser (*class in strlearn.streams*), 22

D

DWM (*class in strlearn.ensembles*), 36

E

ensemble_support_matrix() (*strlearn.ensembles.KUE method*), 40
ensemble_support_matrix() (*strlearn.ensembles.NIE method*), 41
ensemble_support_matrix() (*strlearn.ensembles.OUSE method*), 45
ensemble_support_matrix() (*strlearn.ensembles.ROSE method*), 50
ensemble_support_matrix() (*strlearn.ensembles.StreamingEnsemble method*), 54

F

f1_score() (*in module strlearn.metrics*), 72
fbeta_score() (*in module strlearn.metrics*), 72
fit() (*strlearn.classifiers.ASC method*), 63
fit() (*strlearn.classifiers.Skipper method*), 66
fit() (*strlearn.ensembles.OUSE method*), 45
fit() (*strlearn.ensembles.StreamingEnsemble method*), 54

G

geometric_mean_score_1() (*in module strlearn.metrics*), 73
geometric_mean_score_2() (*in module strlearn.metrics*), 73
get_chunk() (*strlearn.streams.ARFFParser method*), 21
get_chunk() (*strlearn.streams.CSVParser method*), 22
get_chunk() (*strlearn.streams.NPYParser method*), 23
get_chunk() (*strlearn.streams.SemiSyntheticStreamGenerator method*), 24
get_chunk() (*strlearn.streams.StreamGenerator method*), 26

I

is_dry() (*strlearn.streams.ARFFParser method*), 22
is_dry() (*strlearn.streams.CSVParser method*), 23
is_dry() (*strlearn.streams.NPYParser method*), 23

K

KMC (*class in strlearn.ensembles*), 38
KUE (*class in strlearn.ensembles*), 40

M

minority_majority_name() (*strlearn.ensembles.OUSE method*), 45
minority_majority_name() (*strlearn.ensembles.StreamingEnsemble method*), 54
minority_majority_split() (*strlearn.ensembles.OUSE method*), 45
minority_majority_split() (*strlearn.ensembles.StreamingEnsemble method*), 55
module
    strlearn.classifiers, 63
    strlearn.ensembles, 31
    strlearn.evaluators, 27
    strlearn.metrics, 71
    strlearn.streams, 21
    strlearn.utils, 69
msei() (*strlearn.ensembles.StreamingEnsemble method*), 55

mser() (*strlearn.ensembles.StreamingEnsemble method*), 55

N

NIE (*class in strlearn.ensembles*), 41
NPYParser (*class in strlearn.streams*), 23

O

OnlineBagging (*class in strlearn.ensembles*), 47
OOB (*class in strlearn.ensembles*), 43
OUSE (*class in strlearn.ensembles*), 45

P

partial_fit() (*strlearn.classifiers.ASC method*), 63
partial_fit() (*strlearn.ensembles.AUE method*), 31
partial_fit() (*strlearn.ensembles.AWE method*), 33
partial_fit() (*strlearn.ensembles.CDS method*), 35
partial_fit() (*strlearn.ensembles.DWM method*), 36
partial_fit() (*strlearn.ensembles.KMC method*), 38
partial_fit() (*strlearn.ensembles.KUE method*), 40
partial_fit() (*strlearn.ensembles.NIE method*), 41
partial_fit() (*strlearn.ensembles.OnlineBagging method*), 47
partial_fit() (*strlearn.ensembles.OOB method*), 43
partial_fit() (*strlearn.ensembles.OUSE method*), 45
partial_fit() (*strlearn.ensembles.REA method*), 49
partial_fit() (*strlearn.ensembles.ROSE method*), 50
partial_fit() (*strlearn.ensembles.SEA method*), 53
partial_fit() (*strlearn.ensembles.StreamingEnsemble method*), 55
partial_fit() (*strlearn.ensembles.UOB method*), 57
partial_fit() (*strlearn.ensembles.WAE method*), 60
precision() (*in module strlearn.metrics*), 73
predict() (*strlearn.ensembles.OUSE method*), 45
predict() (*strlearn.ensembles.StreamingEnsemble method*), 55
predict_proba() (*strlearn.ensembles.StreamingEnsemble method*), 55
Prequential (*class in strlearn.evaluators*), 27
prior_proba() (*strlearn.ensembles.StreamingEnsemble method*), 55
process() (*strlearn.evaluators.Prequential method*), 27
process() (*strlearn.evaluators.SparseTrainDenseTest method*), 28
process() (*strlearn.evaluators.TestThenTrain method*), 29

R

REA (*class in strlearn.ensembles*), 49
recall() (*in module strlearn.metrics*), 74
reset() (*strlearn.streams.ARFFParser method*), 22
reset() (*strlearn.streams.CSVParser method*), 23
reset() (*strlearn.streams.NPYParser method*), 24
ROSE (*class in strlearn.ensembles*), 50

S

SampleWeightedMetaEstimator (*class in strlearn.classifiers*), 65
save_to_arff() (*strlearn.streams.SemiSyntheticStreamGenerator method*), 25
save_to_arff() (*strlearn.streams.StreamGenerator method*), 26
save_to_csv() (*strlearn.streams.SemiSyntheticStreamGenerator method*), 25
save_to_csv() (*strlearn.streams.StreamGenerator method*), 26
save_to_npy() (*strlearn.streams.SemiSyntheticStreamGenerator method*), 25
save_to_npy() (*strlearn.streams.StreamGenerator method*), 26
scores_to_cummean() (*in module strlearn.utils*), 69
SEA (*class in strlearn.ensembles*), 52
SemiSyntheticStreamGenerator (*class in strlearn.streams*), 24
set_partial_fit_request() (*strlearn.classifiers.ASC method*), 63
set_partial_fit_request() (*strlearn.classifiers.SampleWeightedMetaEstimator method*), 65
set_partial_fit_request() (*strlearn.classifiers.Skipper method*), 66
set_partial_fit_request() (*strlearn.ensembles.AUE method*), 31
set_partial_fit_request() (*strlearn.ensembles.AWE method*), 33
set_partial_fit_request() (*strlearn.ensembles.CDS method*), 35
set_partial_fit_request() (*strlearn.ensembles.DWM method*), 36
set_partial_fit_request() (*strlearn.ensembles.KMC method*), 38
set_partial_fit_request() (*strlearn.ensembles.KUE method*), 40
set_partial_fit_request() (*strlearn.ensembles.NIE method*), 42
set_partial_fit_request() (*strlearn.ensembles.OnlineBagging method*), 47
set_partial_fit_request() (*strlearn.ensembles.OOB method*), 43
set_partial_fit_request() (*strlearn.ensembles.OUSE method*), 45
set_partial_fit_request() (*strlearn.ensembles.REA method*), 49
set_partial_fit_request() (*strlearn.ensembles.ROSE method*), 51

set_partial_fit_request() (*strlearn.ensembles.SEA method*), 53
set_partial_fit_request() (*strlearn.ensembles.StreamingEnsemble method*), 55
set_partial_fit_request() (*strlearn.ensembles.UOB method*), 57
set_partial_fit_request() (*strlearn.ensembles.WAE method*), 60
set_score_request() (*strlearn.classifiers.ASC method*), 64
set_score_request() (*strlearn.classifiers.SampleWeightedMetaEstimator method*), 66
set_score_request() (*strlearn.classifiers.Skipper method*), 67
set_score_request() (*strlearn.ensembles.AUE method*), 32
set_score_request() (*strlearn.ensembles.AWE method*), 34
set_score_request() (*strlearn.ensembles.CDS method*), 36
set_score_request() (*strlearn.ensembles.DWM method*), 37
set_score_request() (*strlearn.ensembles.KMC method*), 39
set_score_request() (*strlearn.ensembles.KUE method*), 41
set_score_request() (*strlearn.ensembles.NIE method*), 42
set_score_request() (*strlearn.ensembles.OnlineBagging method*), 48
set_score_request() (*strlearn.ensembles.OOB method*), 44
set_score_request() (*strlearn.ensembles.OUSE method*), 46
set_score_request() (*strlearn.ensembles.REA method*), 50
set_score_request() (*strlearn.ensembles.ROSE method*), 51
set_score_request() (*strlearn.ensembles.SEA method*), 54
set_score_request() (*strlearn.ensembles.StreamingEnsemble method*), 56
set_score_request() (*strlearn.ensembles.UOB method*), 58
set_score_request() (*strlearn.ensembles.WAE method*), 61
Skipper (*class in strlearn.classifiers*), 66
SparseTrainDenseTest (*class in strlearn.evaluators*), 28
specificity() (*in module strlearn.metrics*), 74
StreamGenerator (*class in strlearn.streams*), 25
StreamingEnsemble (*class in strlearn.ensembles*), 54
strlearn.classifiers
    module, 63
strlearn.ensembles
    module, 31
strlearn.evaluators
    module, 27
strlearn.metrics
    module, 71
strlearn.streams
    module, 21
strlearn.utils
    module, 69

T

TestThenTrain (*class in strlearn.evaluators*), 28

U

UOB (*class in strlearn.ensembles*), 57

W

WAE (*class in strlearn.ensembles*), 59