`imblearn.under_sampling`.NearMiss

class **`imblearn.under_sampling.NearMiss`**(*sampling_strategy='auto', return_indices=False, random_state=None, version=1, n_neighbors=3, n_neighbors_ver3=3, n_jobs=1, ratio=None*) [source]

[source]

Class to perform under-sampling based on NearMiss methods.

Read more in the User Guide.

Parameters:

sampling_strategy:float, str, dict, callable, (default='auto')

Sampling information to sample the data set.

- When `float`, it corresponds to the desired ratio of the number of samples in the majority class over the number of samples in the minority class after resampling. Therefore, the ratio is expressed as $\alpha_{us} = N_{rM}/N_{m}$ where N_{rM} and N_{m} are the number of samples in the majority class after resampling and the number of samples in the minority class, respectively.

  Warning

  `float` is only available for **binary** classification. An error is raised for multi-class classification.

- When `str`, specify the class targeted by the resampling. The number of samples in the different classes will be equalized. Possible choices are:

  `'majority'`: resample only the majority class;

  `'not minority'`: resample all classes but the minority class;

  `'not majority'`: resample all classes but the majority class;

  `'all'`: resample all classes;

  `'auto'`: equivalent to `'not minority'`.

- When `dict`, the keys correspond to the targeted classes. The values correspond to the desired number of samples for each targeted class.

- When callable, function taking `y` and returns a `dict`. The keys correspond to the targeted classes. The values correspond to the desired number of samples for each class.

return_indices:bool, optional (default=False)

Whether or not to return the indices of the samples randomly selected from the majority class.

Deprecated since version 0.4: `return_indices` is deprecated. Use the attribute `sample_indices_` instead.

random_state:int, RandomState instance or None, optional (default=None)

Control the randomization of the algorithm.

- If int, `random_state` is the seed used by the random number generator;
- If `RandomState` instance, random_state is the random number generator;
- If `None`, the random number generator is the `RandomState` instance used by `np.random`.

Deprecated since version 0.4: `random_state` is deprecated in 0.4 and will be removed in 0.6.

version:int, optional (default=1)

Version of the NearMiss to use. Possible values are 1, 2 or 3.

n_neighbors:int or object, optional (default=3)

If `int`, size of the neighbourhood to consider to compute the average distance to the minority point samples. If object, an estimator that inherits from `sklearn.neighbors.base.KNeighborsMixin` that will be used to find the k_neighbors.

n_neighbors_ver3:int or object, optional (default=3)

If `int`, NearMiss-3 algorithm start by a phase of re-sampling. This parameter correspond to the number of neighbours selected create the subset in which the selection will be performed. If object, an estimator that inherits from `sklearn.neighbors.base.KNeighborsMixin` that will be used to find the k_neighbors.

n_jobs:int, optional (default=1)

The number of threads to open if possible.

ratio:str, dict, or callable

Deprecated since version 0.4: Use the parameter `sampling_strategy` instead. It will be removed in 0.6.

Notes

The methods are based on [1].

Supports multi-class resampling.

References

[1] (1, 2) I. Mani, I. Zhang. "kNN approach to unbalanced data distributions: a case study involving information extraction," In Proceedings of workshop on learning from imbalanced datasets, 2003.

Examples

```
>>> from collections import Counter
>>> from sklearn.datasets import make_classification
>>> from imblearn.under_sampling import NearMiss # doctest: +NORMALIZE_WHITESPACE
>>> X, y = make_classification(n_classes=2, class_sep=2,
... weights=[0.1, 0.9], n_informative=3, n_redundant=1, flip_y=0,
... n_features=20, n_clusters_per_class=1, n_samples=1000, random_state=10)
>>> print('Original dataset shape %s' % Counter(y))
Original dataset shape Counter({1: 900, 0: 100})
>>> nm = NearMiss()
>>> X_res, y_res = nm.fit_resample(X, y)
>>> print('Resampled dataset shape %s' % Counter(y_res))
Resampled dataset shape Counter({0: 100, 1: 100})
```

Attributes:

sample_indices_:ndarray, shape (n_new_samples)

Indices of the samples selected.

New in version 0.4: `sample_indices_` used instead of `return_indices=True`.

`__init__`(*sampling_strategy='auto', return_indices=False, random_state=None, version=1, n_neighbors=3, n_neighbors_ver3=3, n_jobs=1, ratio=None*) [source] [source]

Initialize self. See help(type(self)) for accurate signature.

`fit`(*X, y*) [source]

Check inputs and statistics of the sampler.

You should use `fit_resample` in all cases.

Parameters:

X:{array-like, sparse matrix}, shape (n_samples, n_features)

Data array.

y:array-like, shape (n_samples,)

Target array.

Returns:

self:object

Return the instance itself.

`fit_resample`(*X, y*) [source]

Resample the dataset.

Parameters:

X:{array-like, sparse matrix}, shape (n_samples, n_features)
Matrix containing the data which have to be sampled.

y:array-like, shape (n_samples,)
Corresponding label for each sample in X.

Returns:

X_resampled:{array-like, sparse matrix}, shape (n_samples_new, n_features)
The array containing the resampled data.

y_resampled:array-like, shape (n_samples_new,)
The corresponding label of *X_resampled*.

`fit_sample`(*X*, *y*) [source]

Resample the dataset.

Parameters:

X:{array-like, sparse matrix}, shape (n_samples, n_features)
Matrix containing the data which have to be sampled.

y:array-like, shape (n_samples,)
Corresponding label for each sample in X.

Returns:

X_resampled:{array-like, sparse matrix}, shape (n_samples_new, n_features)
The array containing the resampled data.

y_resampled:array-like, shape (n_samples_new,)
The corresponding label of *X_resampled*.

`get_params`(*deep=True*) [source]

Get parameters for this estimator.

Parameters:

deep:boolean, optional
If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns:

params:mapping of string to any
Parameter names mapped to their values.

`set_params`(***params*) [source]

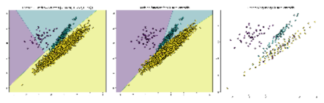
Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The latter have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Returns: self

Examples using `imblearn.under_sampling.NearMiss`

Multiclass classification with under-sampling



Comparison of the different under-sampling algorithms