**What is Sampling?**

Let’s start by formally defining what sampling is.

Sampling is a method that allows us to get information about the population based on the statistics from a subset of the population (sample), without having to investigate every individual.

**Why do we need Sampling?**

Sampling is done to draw conclusions about populations from samples, and it enables us to determine a population’s characteristics by directly observing only a portion (or sample) of the population.

* Selecting a sample requires less time than selecting every item in a population
* Sample selection is a cost-efficient method
* Analysis of the sample is less cumbersome and more practical than an analysis of the entire population

This bias in the training dataset can influence many machine learning algorithms, leading some to ignore the minority class entirely. This is a problem as it is typically the minority class on which predictions are most important.

One approach to addressing the problem of class imbalance is to randomly resample the training dataset. The two main approaches to randomly resampling an imbalanced dataset are to delete examples from the majority class, called undersampling, and to duplicate examples from the minority class, called oversampling.

Both approaches can be repeated until the desired class distribution is achieved in the training dataset, such as an equal split across the classes.

They are referred to as “naive resampling” methods because they assume nothing about the data and no heuristics are used. This makes them simple to implement and fast to execute, which is desirable for very large and complex datasets.

Both techniques can be used for two-class (binary) classification problems and multi-class classification problems with one or more majority or minority classes.

Importantly, the change to the class distribution is only applied to the training dataset. The intent is to influence the fit of the models. The resampling is not applied to the test or holdout dataset used to evaluate the performance of a model.

Generally, these naive methods can be effective, although that depends on the specifics of the dataset and models involved.

from numpy import mean

from sklearn.datasets import make\_classification

from imblearn.pipeline import Pipeline

from imblearn.over\_sampling import RandomOverSample

**Random Oversampling Imbalanced Datasets**

Random oversampling involves randomly duplicating examples from the minority class and adding them to the training dataset.

Examples from the training dataset are selected randomly with replacement. This means that examples from the minority class can be chosen and added to the new “*more balanced*” training dataset multiple times; they are selected from the original training dataset, added to the new training dataset, and then returned or “*replaced*” in the original dataset, allowing them to be selected again.

This technique can be effective for those machine learning algorithms that are affected by a skewed distribution and where multiple duplicate examples for a given class can influence the fit of the model. This might include algorithms that iteratively learn coefficients, like artificial neural networks that use stochastic gradient descent. It can also affect models that seek good splits of the data, such as support vector machines and decision trees.

It might be useful to tune the target class distribution. In some cases, seeking a balanced distribution for a severely imbalanced dataset can cause affected algorithms to overfit the minority class, leading to increased generalization error. The effect can be better performance on the training dataset, but worse performance on the holdout or test dataset.

*… the random oversampling may increase the likelihood of occurring overfitting, since it makes exact copies of the minority class examples. In this way, a symbolic classifier, for instance, might construct rules that are apparently accurate, but actually cover one replicated example.*

— Page 83, [Learning from Imbalanced Data Sets](https://amzn.to/307Xlva), 2018.

As such, to gain insight into the impact of the method, it is a good idea to monitor the performance on both train and test datasets after oversampling and compare the results to the same algorithm on the original dataset.

The increase in the number of examples for the minority class, especially if the class skew was severe, can also result in a marked increase in the computational cost when fitting the model, especially considering the model is seeing the same examples in the training dataset again and again.

*… in random over-sampling, a random set of copies of minority class examples is added to the data. This may increase the likelihood of overfitting, specially for higher over-sampling rates. Moreover, it may decrease the classifier performance and increase the computational effort.*

— [A Survey of Predictive Modelling under Imbalanced Distributions](https://arxiv.org/abs/1505.01658), 2015.

Random oversampling can be implemented using the [RandomOverSampler class](https://imbalanced-learn.org/stable/generated/imblearn.over_sampling.RandomOverSampler.html).

The class can be defined and takes a *sampling\_strategy* argument that can be set to “*minority*” to automatically balance the minority class with majority class or classes.

For example:

|  |  |
| --- | --- |
| 1  2  3 | ...  # define oversampling strategy  oversample = RandomOverSampler(sampling\_strategy='minority') |

This means that if the majority class had 1,000 examples and the minority class had 100, this strategy would oversampling the minority class so that it has 1,000 examples.

A floating point value can be specified to indicate the ratio of minority class majority examples in the transformed dataset. For example:

|  |  |
| --- | --- |
| 1  2  3 | ...  # define oversampling strategy  oversample = RandomOverSampler(sampling\_strategy=0.5) |

This would ensure that the minority class was oversampled to have half the number of examples as the majority class, for binary classification problems. This means that if the majority class had 1,000 examples and the minority class had 100, the transformed dataset would have 500 examples of the minority class.

The class is like a scikit-learn transform object in that it is fit on a dataset, then used to generate a new or transformed dataset. Unlike the scikit-learn transforms, it will change the number of examples in the dataset, not just the values (like a scaler) or number of features (like a projection).

For example, it can be fit and applied in one step by calling the *fit\_sample()* function:

|  |  |
| --- | --- |
| 1  2  3 | ...  # fit and apply the transform  X\_over, y\_over = oversample.fit\_resample(X, y) |

We can demonstrate this on a simple synthetic binary classification problem with a 1:100 class imbalance.

|  |  |
| --- | --- |
| 1  2  3 | ...  # define dataset  X, y = make\_classification(n\_samples=10000, weights=[0.99], flip\_y=0) |

The complete example of defining the dataset and performing random oversampling to balance the class distribution is listed below.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | # example of random oversampling to balance the class distribution  from collections import Counter  from sklearn.datasets import make\_classification  from imblearn.over\_sampling import RandomOverSampler  # define dataset  X, y = make\_classification(n\_samples=10000, weights=[0.99], flip\_y=0)  # summarize class distribution  print(Counter(y))  # define oversampling strategy  oversample = RandomOverSampler(sampling\_strategy='minority')  # fit and apply the transform  X\_over, y\_over = oversample.fit\_resample(X, y)  # summarize class distribution  print(Counter(y\_over)) |

Running the example first creates the dataset, then summarizes the class distribution. We can see that there are nearly 10K examples in the majority class and 100 examples in the minority class.

Then the random oversample transform is defined to balance the minority class, then fit and applied to the dataset. The class distribution for the transformed dataset is reported showing that now the minority class has the same number of examples as the majority class.

|  |  |
| --- | --- |
| 1  2 | Counter({0: 9900, 1: 100})  Counter({0: 9900, 1: 9900}) |

This transform can be used as part of a *Pipeline* to ensure that it is only applied to the training dataset as part of each split in a k-fold cross validation.

A traditional scikit-learn [Pipeline](https://scikit-learn.org/stable/modules/generated/sklearn.pipeline.Pipeline.html) cannot be used; instead, a [Pipeline from the imbalanced-learn library](https://imbalanced-learn.org/stable/generated/imblearn.pipeline.Pipeline.html) can be used. For example:

|  |  |
| --- | --- |
| 1  2  3  4 | ...  # pipeline  steps = [('over', RandomOverSampler()), ('model', DecisionTreeClassifier())]  pipeline = Pipeline(steps=steps) |

The example below provides a complete example of evaluating a decision tree on an imbalanced dataset with a 1:100 class distribution.

The model is evaluated using repeated 10-fold cross-validation with three repeats, and the oversampling is performed on the training dataset within each fold separately, ensuring that there is no data leakage as might occur if the oversampling was performed prior to the cross-validation.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18 | # example of evaluating a decision tree with random oversampling  from numpy import mean  from sklearn.datasets import make\_classification  from sklearn.model\_selection import cross\_val\_score  from sklearn.model\_selection import RepeatedStratifiedKFold  from sklearn.tree import DecisionTreeClassifier  from imblearn.pipeline import Pipeline  from imblearn.over\_sampling import RandomOverSampler  # define dataset  X, y = make\_classification(n\_samples=10000, weights=[0.99], flip\_y=0)  # define pipeline  steps = [('over', RandomOverSampler()), ('model', DecisionTreeClassifier())]  pipeline = Pipeline(steps=steps)  # evaluate pipeline  cv = RepeatedStratifiedKFold(n\_splits=10, n\_repeats=3, random\_state=1)  scores = cross\_val\_score(pipeline, X, y, scoring='f1\_micro', cv=cv, n\_jobs=-1)  score = mean(scores)  print('F1 Score: %.3f' % score) |

Running the example evaluates the decision tree model on the imbalanced dataset with oversampling.

The chosen model and resampling configuration are arbitrary, designed to provide a template that you can use to test undersampling with your dataset and learning algorithm, rather than optimally solve the synthetic dataset.

**Note**: Your [results may vary](https://machinelearningmastery.com/different-results-each-time-in-machine-learning/) given the stochastic nature of the algorithm or evaluation procedure, or differences in numerical precision. Consider running the example a few times and compare the average outcome.

The default oversampling strategy is used, which balances the minority classes with the majority class. The F1 score averaged across each fold and each repeat is reported.

|  |  |
| --- | --- |
| 1 | F1 Score: 0.990 |

Now that we are familiar with oversampling, let’s take a look at undersampling.

**Random Undersampling Imbalanced Datasets**

Random undersampling involves randomly selecting examples from the majority class to delete from the training dataset.

This has the effect of reducing the number of examples in the majority class in the transformed version of the training dataset. This process can be repeated until the desired class distribution is achieved, such as an equal number of examples for each class.

This approach may be more suitable for those datasets where there is a class imbalance although a sufficient number of examples in the minority class, such a useful model can be fit.

A limitation of undersampling is that examples from the majority class are deleted that may be useful, important, or perhaps critical to fitting a robust decision boundary. Given that examples are deleted randomly, there is no way to detect or preserve “*good*” or more information-rich examples from the majority class.

*… in random under-sampling (potentially), vast quantities of data are discarded. […] This can be highly problematic, as the loss of such data can make the decision boundary between minority and majority instances harder to learn, resulting in a loss in classification performance.*

— Page 45, [Imbalanced Learning: Foundations, Algorithms, and Applications](https://amzn.to/32K9K6d), 2013

The random undersampling technique can be implemented using the [RandomUnderSampler imbalanced-learn class](https://imbalanced-learn.org/stable/generated/imblearn.under_sampling.RandomUnderSampler.html).

The class can be used just like the *RandomOverSampler* class in the previous section, except the strategies impact the majority class instead of the minority class. For example, setting the *sampling\_strategy* argument to “*majority*” will undersample the majority class determined by the class with the largest number of examples.

|  |  |
| --- | --- |
| 1  2  3 | ...  # define undersample strategy  undersample = RandomUnderSampler(sampling\_strategy='majority') |

For example, a dataset with 1,000 examples in the majority class and 100 examples in the minority class will be undersampled such that both classes would have 100 examples in the transformed training dataset.

We can also set the *sampling\_strategy* argument to a floating point value which will be a percentage relative to the minority class, specifically the number of examples in the minority class divided by the number of examples in the majority class. For example, if we set *sampling\_strategy* to 0.5 in an imbalanced data dataset with 1,000 examples in the majority class and 100 examples in the minority class, then there would be 200 examples for the majority class in the transformed dataset (or 100/200 = 0.5).

|  |  |
| --- | --- |
| 1  2  3 | ...  # define undersample strategy  undersample = RandomUnderSampler(sampling\_strategy=0.5) |

This might be preferred to ensure that the resulting dataset is both large enough to fit a reasonable model, and that not too much useful information from the majority class is discarded.

*In random under-sampling, one might attempt to create a balanced class distribution by selecting 90 majority class instances at random to be removed. The resulting dataset will then consist of 20 instances: 10 (randomly remaining) majority class instances and (the original) 10 minority class instances.*

— Page 45, [Imbalanced Learning: Foundations, Algorithms, and Applications](https://amzn.to/32K9K6d), 2013

The transform can then be fit and applied to a dataset in one step by calling the *fit\_resample()* function and passing the untransformed dataset as arguments.

|  |  |
| --- | --- |
| 1  2  3 | ...  # fit and apply the transform  X\_over, y\_over = undersample.fit\_resample(X, y) |

We can demonstrate this on a dataset with a 1:100 class imbalance.

The complete example is listed below.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | # example of random undersampling to balance the class distribution  from collections import Counter  from sklearn.datasets import make\_classification  from imblearn.under\_sampling import RandomUnderSampler  # define dataset  X, y = make\_classification(n\_samples=10000, weights=[0.99], flip\_y=0)  # summarize class distribution  print(Counter(y))  # define undersample strategy  undersample = RandomUnderSampler(sampling\_strategy='majority')  # fit and apply the transform  X\_over, y\_over = undersample.fit\_resample(X, y)  # summarize class distribution  print(Counter(y\_over)) |

Running the example first creates the dataset and reports the imbalanced class distribution.

The transform is fit and applied on the dataset and the new class distribution is reported. We can see that that majority class is undersampled to have the same number of examples as the minority class.

Judgment and empirical results will have to be used as to whether a training dataset with just 200 examples would be sufficient to train a model.

|  |  |
| --- | --- |
| 1  2 | Counter({0: 9900, 1: 100})  Counter({0: 100, 1: 100}) |

This undersampling transform can also be used in a Pipeline, like the oversampling transform from the previous section.

This allows the transform to be applied to the training dataset only using evaluation schemes such as k-fold cross-validation, avoiding any data leakage in the evaluation of a model.

|  |  |
| --- | --- |
| 1  2  3  4 | ...  # define pipeline  steps = [('under', RandomUnderSampler()), ('model', DecisionTreeClassifier())]  pipeline = Pipeline(steps=steps) |

We can define an example of fitting a decision tree on an imbalanced classification dataset with the undersampling transform applied to the training dataset on each split of a repeated 10-fold cross-validation.

The complete example is listed below.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18 | # example of evaluating a decision tree with random undersampling  from numpy import mean  from sklearn.datasets import make\_classification  from sklearn.model\_selection import cross\_val\_score  from sklearn.model\_selection import RepeatedStratifiedKFold  from sklearn.tree import DecisionTreeClassifier  from imblearn.pipeline import Pipeline  from imblearn.under\_sampling import RandomUnderSampler  # define dataset  X, y = make\_classification(n\_samples=10000, weights=[0.99], flip\_y=0)  # define pipeline  steps = [('under', RandomUnderSampler()), ('model', DecisionTreeClassifier())]  pipeline = Pipeline(steps=steps)  # evaluate pipeline  cv = RepeatedStratifiedKFold(n\_splits=10, n\_repeats=3, random\_state=1)  scores = cross\_val\_score(pipeline, X, y, scoring='f1\_micro', cv=cv, n\_jobs=-1)  score = mean(scores)  print('F1 Score: %.3f' % score) |

Running the example evaluates the decision tree model on the imbalanced dataset with undersampling.

The chosen model and resampling configuration are arbitrary, designed to provide a template that you can use to test undersampling with your dataset and learning algorithm rather than optimally solve the synthetic dataset.

**Note**: Your [results may vary](https://machinelearningmastery.com/different-results-each-time-in-machine-learning/) given the stochastic nature of the algorithm or evaluation procedure, or differences in numerical precision. Consider running the example a few times and compare the average outcome.

The default undersampling strategy is used, which balances the majority classes with the minority class. The F1 score averaged across each fold and each repeat is reported.

|  |  |
| --- | --- |
| 1 | F1 Score: 0.889 |