Machine Learning Mastery
Making Developers Awesome at Machine Learning

SMOTE for Imbalanced Classification with Python

by **Jason Brownlee** on January 17, 2020 in **Imbalanced Classification**

Last Updated on March 17, 2021

Imbalanced classification involves developing predictive models on classification datasets that have a severe class imbalance.

The challenge of working with imbalanced datasets is that most machine learning techniques will ignore, and in turn have poor performance on, the minority class, although typically it is performance on the minority class that is most important.

One approach to addressing imbalanced datasets is to oversample the minority class. The simplest approach involves duplicating examples in the minority class, although these examples don't add any new information to the model. Instead, new examples can be synthesized from the existing examples. This is a type of data augmentation for the minority class and is referred to as the **Synthetic Minority Oversampling Technique**, or **SMOTE** for short.

In this tutorial, you will discover the SMOTE for oversampling imbalanced classification datasets.

After completing this tutorial, you will know:

- How the SMOTE synthesizes new examples for the minority class.
- How to correctly fit and evaluate machine learning models on SMOTE-transformed training datasets.
- How to use extensions of the SMOTE that generate synthetic examples along the class decision boundary.

Kick-start your project with my new book Imbalanced Classification with Python, including *step-by-step tutorials* and the *Python source code* files for all examples.

Let's get started.

- **Updated Jan/2021**: Updated links for API documentation.

SMOTE Oversampling for Imbalanced Classification with Python
Photo by Victor U, some rights reserved.

Tutorial Overview

This tutorial is divided into five parts; they are:

1. Synthetic Minority Oversampling Technique
2. Imbalanced-Learn Library
3. SMOTE for Balancing Data
4. SMOTE for Classification
5. SMOTE With Selective Synthetic Sample Generation
    1. Borderline-SMOTE
    2. Borderline-SMOTE SVM
    3. Adaptive Synthetic Sampling (ADASYN)

Synthetic Minority Oversampling Technique

A problem with imbalanced classification is that there are too few examples of the minority class for a model to effectively learn the decision boundary.

One way to solve this problem is to oversample the examples in the minority class. This can be achieved by simply duplicating examples from the minority class in the training dataset prior to fitting a model. This can balance the class distribution but does not provide any additional information to the model.

An improvement on duplicating examples from the minority class is to synthesize new examples from the minority class. This is a type of data augmentation for tabular data and can be very effective.

Perhaps the most widely used approach to synthesizing new examples is called the **Synthetic Minority Oversampling TEchnique**, or SMOTE for short. This technique was described by Nitesh Chawla, et al. in their 2002 paper named for the technique titled "SMOTE: Synthetic Minority Over-sampling Technique."

SMOTE works by selecting examples that are close in the feature space, drawing a line between the examples in the feature space and drawing a new sample at a point along that line.

Specifically, a random example from the minority class is first chosen. Then *k* of the nearest neighbors for that example are found (typically *k=5*). A randomly selected neighbor is chosen and a synthetic example is created at a randomly selected point between the two examples in feature space.

> *... SMOTE first selects a minority class instance a at random and finds its k nearest minority class neighbors. The synthetic instance is then created by choosing one of the k nearest neighbors b at random and connecting a and b to form a line segment in the feature space. The synthetic instances are generated as a convex combination of the two chosen instances a and b.*

— Page 47, Imbalanced Learning: Foundations, Algorithms, and Applications, 2013.

This procedure can be used to create as many synthetic examples for the minority class as are required. As described in the paper, it suggests first using random undersampling to trim the number of examples in the majority class, then use SMOTE to oversample the minority class to balance the class distribution.

> *The combination of SMOTE and under-sampling performs better than plain under-sampling.*

— SMOTE: Synthetic Minority Over-sampling Technique, 2011.

The approach is effective because new synthetic examples from the minority class are created that are plausible, that is, are relatively close in feature space to existing examples from the minority class.

> *Our method of synthetic over-sampling works to cause the classifier to build larger decision regions that contain nearby minority class points.*

— SMOTE: Synthetic Minority Over-sampling Technique, 2011.

A general downside of the approach is that synthetic examples are created without considering the majority class, possibly resulting in ambiguous examples if there is a strong overlap for the classes.

Now that we are familiar with the technique, let's look at a worked example for an imbalanced classification problem.

Imbalanced-Learn Library

In these examples, we will use the implementations provided by the imbalanced-learn Python library, which can be installed via pip as follows:

```
sudo pip install imbalanced-learn
```

You can confirm that the installation was successful by printing the version of the installed library:

```
# check version number
import imblearn
print(imblearn.__version__)
```

Running the example will print the version number of the installed library; for example:

```
0.5.0
```


SMOTE for Balancing Data

In this section, we will develop an intuition for the SMOTE by applying it to an imbalanced binary classification problem.

First, we can use the make_classification() scikit-learn function to create a synthetic binary classification dataset with 10,000 examples and a 1:100 class distribution.

```
...
# define dataset
X, y = make_classification(n_samples=10000, n_features=2, n_redundant=0,
 n_clusters_per_class=1, weights=[0.99], flip_y=0, random_state=1)
```

We can use the Counter object to summarize the number of examples in each class to confirm the dataset was created correctly.

```
...
# summarize class distribution
counter = Counter(y)
print(counter)
```

Finally, we can create a scatter plot of the dataset and color the examples for each class a different color to clearly see the spatial nature of the class imbalance.

```
...
# scatter plot of examples by class label
for label, _ in counter.items():
 row_ix = where(y == label)[0]
 pyplot.scatter(X[row_ix, 0], X[row_ix, 1], label=str(label))
pyplot.legend()
pyplot.show()
```

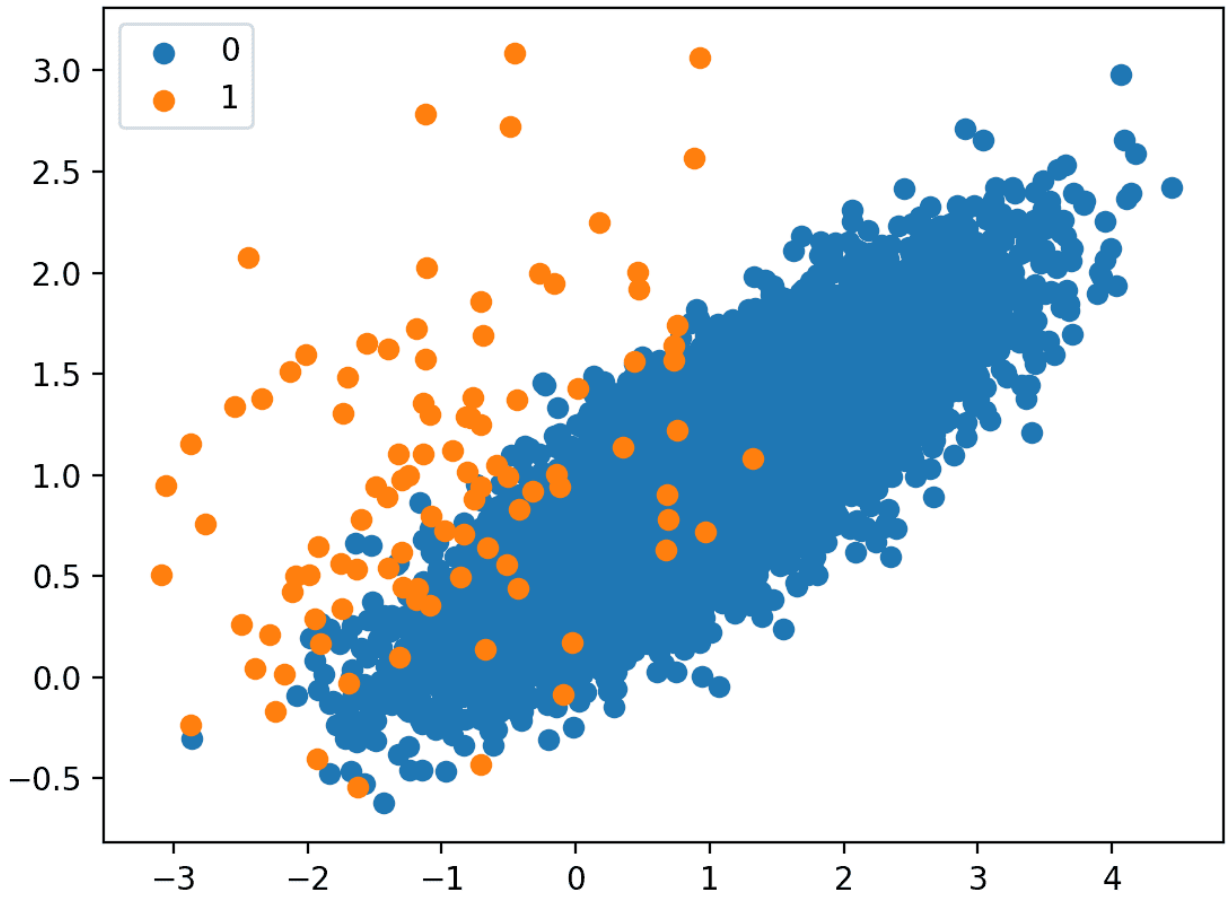
Tying this all together, the complete example of generating and plotting a synthetic binary classification problem is listed below.

```
# Generate and plot a synthetic imbalanced classification dataset
from collections import Counter
from sklearn.datasets import make_classification
from matplotlib import pyplot
from numpy import where
# define dataset
X, y = make_classification(n_samples=10000, n_features=2, n_redundant=0,
 n_clusters_per_class=1, weights=[0.99], flip_y=0, random_state=1)
# summarize class distribution
counter = Counter(y)
print(counter)
# scatter plot of examples by class label
for label, _ in counter.items():
 row_ix = where(y == label)[0]
 pyplot.scatter(X[row_ix, 0], X[row_ix, 1], label=str(label))
pyplot.legend()
pyplot.show()
```

Running the example first summarizes the class distribution, confirms the 1:100 ratio, in this case with about 9,900 examples in the majority class and 100 in the minority class.

```
Counter({0: 9900, 1: 100})
```

A scatter plot of the dataset is created showing the large mass of points that belong to the majority class (blue) and a small number of points spread out for the minority class (orange). We can see some measure of overlap between the two classes.



Scatter Plot of Imbalanced Binary Classification Problem

Next, we can oversample the minority class using SMOTE and plot the transformed dataset.

We can use the SMOTE implementation provided by the imbalanced-learn Python library in the SMOTE class.

The SMOTE class acts like a data transform object from scikit-learn in that it must be defined and configured, fit on a dataset, then applied to create a new transformed version of the dataset.

For example, we can define a SMOTE instance with default parameters that will balance the minority class and then fit and apply it in one step to create a transformed version of our dataset.

```
...
# transform the dataset
oversample = SMOTE()
X, y = oversample.fit_resample(X, y)
```

Once transformed, we can summarize the class distribution of the new transformed dataset, which would expect to now be balanced through the creation of many new synthetic examples in the minority class.

```
...
# summarize the new class distribution
counter = Counter(y)
print(counter)
```

A scatter plot of the transformed dataset can also be created and we would expect to see many more examples for the minority class on lines between the original examples in the minority class.

Tying this together, the complete examples of applying SMOTE to the synthetic dataset and then summarizing and plotting the transformed result is listed below.

```
# Oversample and plot imbalanced dataset with SMOTE
from collections import Counter
from sklearn.datasets import make_classification
from imblearn.over_sampling import SMOTE
from matplotlib import pyplot
from numpy import where
# define dataset
X, y = make_classification(n_samples=10000, n_features=2, n_redundant=0,
 n_clusters_per_class=1, weights=[0.99], flip_y=0, random_state=1)
# summarize class distribution
counter = Counter(y)
print(counter)
# transform the dataset
oversample = SMOTE()
X, y = oversample.fit_resample(X, y)
# summarize the new class distribution
counter = Counter(y)
print(counter)
# scatter plot of examples by class label
for label, _ in counter.items():
 row_ix = where(y == label)[0]
 pyplot.scatter(X[row_ix, 0], X[row_ix, 1], label=str(label))
pyplot.legend()
pyplot.show()
```

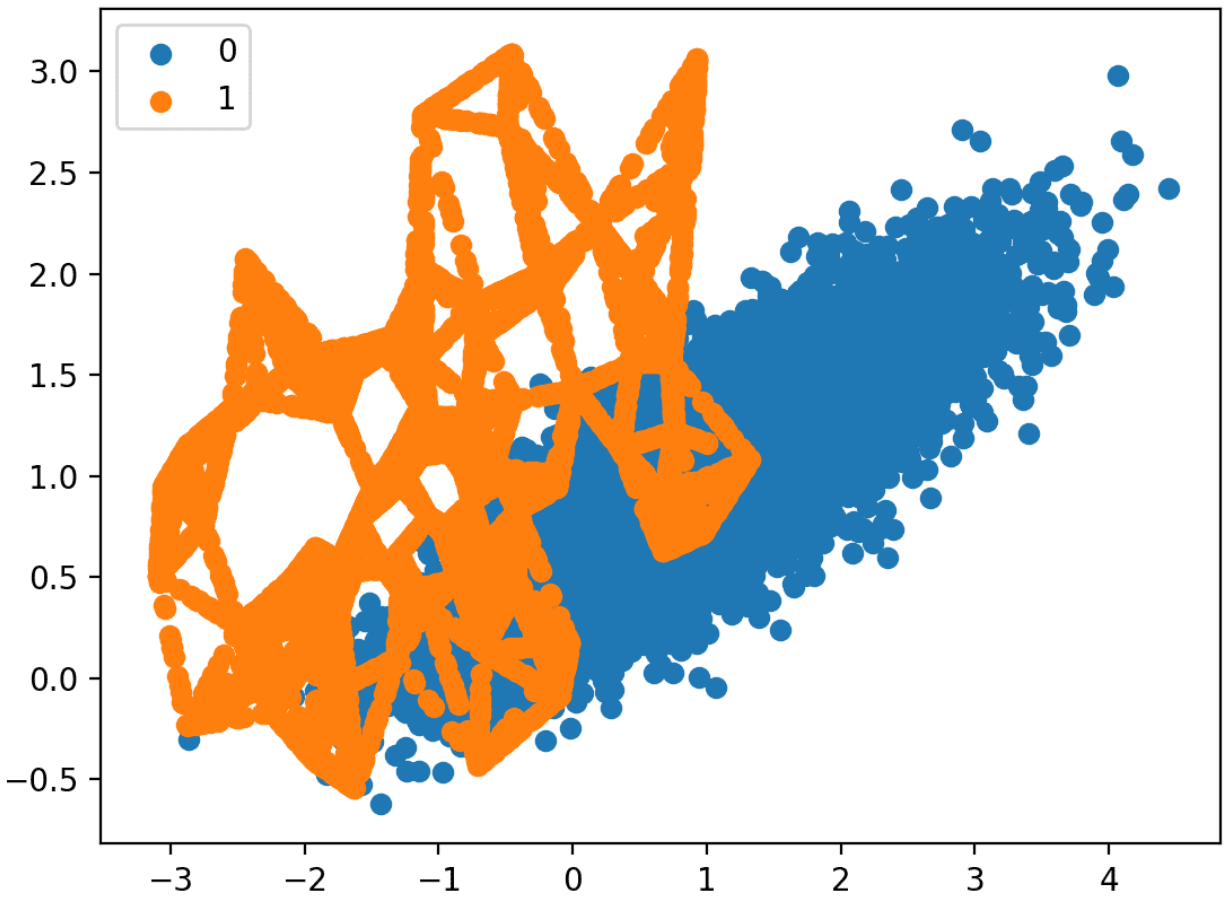
Running the example first creates the dataset and summarizes the class distribution, showing the 1:100 ratio.

Then the dataset is transformed using the SMOTE and the new class distribution is summarized, showing a balanced distribution now with 9,900 examples in the minority class.

```
Counter({0: 9900, 1: 100})
Counter({0: 9900, 1: 9900})
```

Finally, a scatter plot of the transformed dataset is created.

It shows many more examples in the minority class created along the lines between the original examples in the minority class.



Scatter Plot of Imbalanced Binary Classification Problem Transformed by SMOTE

The original paper on SMOTE suggested combining SMOTE with random undersampling of the majority class.

The imbalanced-learn library supports random undersampling via the RandomUnderSampler class.

We can update the example to first oversample the minority class to have 10 percent the number of examples of the majority class (e.g. about 1,000), then use random undersampling to reduce the number of examples in the majority class to have 50 percent more than the minority class (e.g. about 2,000).

To implement this, we can specify the desired ratios as arguments to the SMOTE and *RandomUnderSampler* classes; for example:

```
...
over = SMOTE(sampling_strategy=0.1)
under = RandomUnderSampler(sampling_strategy=0.5)
```

We can then chain these two transforms together into a Pipeline.

The Pipeline can then be applied to a dataset, performing each transformation in turn and returning a final dataset with the accumulation of the transform applied to it, in this case oversampling followed by undersampling.

```
...
steps = [('o', over), ('u', under)]
pipeline = Pipeline(steps=steps)
```

The pipeline can then be fit and applied to our dataset just like a single transform:

```
...
# transform the dataset
X, y = pipeline.fit_resample(X, y)
```

We can then summarize and plot the resulting dataset.

We would expect some SMOTE oversampling of the minority class, although not as much as before where the dataset was balanced. We also expect fewer examples in the majority class via random undersampling.

Tying this all together, the complete example is listed below.

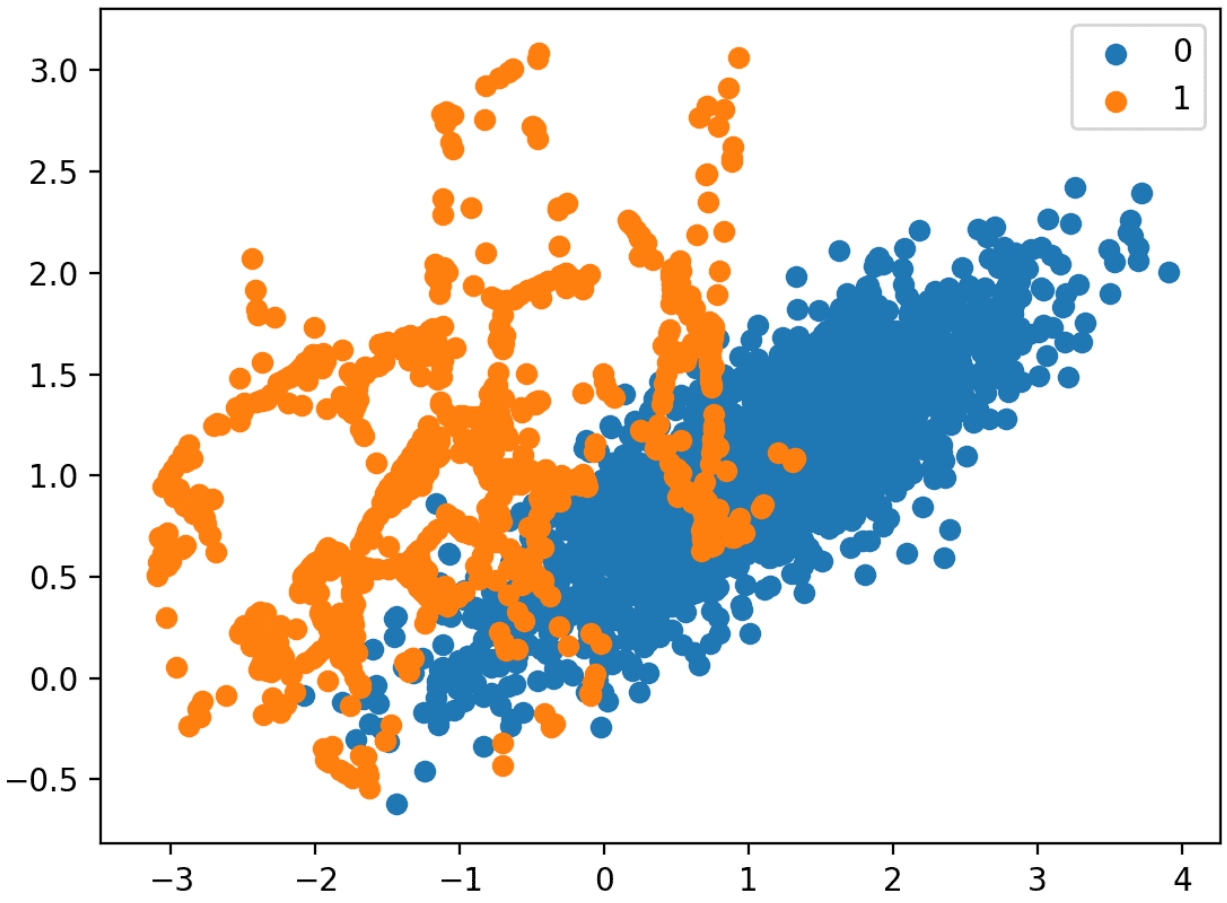
```
# Oversample with SMOTE and random undersample for imbalanced dataset
from collections import Counter
from sklearn.datasets import make_classification
from imblearn.over_sampling import SMOTE
from imblearn.under_sampling import RandomUnderSampler
from imblearn.pipeline import Pipeline
from matplotlib import pyplot
from numpy import where
# define dataset
X, y = make_classification(n_samples=10000, n_features=2, n_redundant=0,
 n_clusters_per_class=1, weights=[0.99], flip_y=0, random_state=1)
# summarize class distribution
counter = Counter(y)
print(counter)
# define pipeline
over = SMOTE(sampling_strategy=0.1)
under = RandomUnderSampler(sampling_strategy=0.5)
steps = [('o', over), ('u', under)]
pipeline = Pipeline(steps=steps)
# transform the dataset
X, y = pipeline.fit_resample(X, y)
# summarize the new class distribution
counter = Counter(y)
print(counter)
# scatter plot of examples by class label
for label, _ in counter.items():
 row_ix = where(y == label)[0]
 pyplot.scatter(X[row_ix, 0], X[row_ix, 1], label=str(label))
pyplot.legend()
pyplot.show()
```

Running the example first creates the dataset and summarizes the class distribution.

Next, the dataset is transformed, first by oversampling the minority class, then undersampling the majority class. The final class distribution after this sequence of transforms matches our expectations with a 1:2 ratio or about 2,000 examples in the majority class and about 1,000 examples in the minority class.

```
Counter({0: 9900, 1: 100})
Counter({0: 1980, 1: 990})
```

Finally, a scatter plot of the transformed dataset is created, showing the oversampled minority class and the undersampled majority class.



Scatter Plot of Imbalanced Dataset Transformed by SMOTE and Random Undersampling

Now that we are familiar with transforming imbalanced datasets, let's look at using SMOTE when fitting and evaluating classification models.

SMOTE for Classification

In this section, we will look at how we can use SMOTE as a data preparation method when fitting and evaluating machine learning algorithms in scikit-learn.

First, we use our binary classification dataset from the previous section then fit and evaluate a decision tree algorithm.

The algorithm is defined with any required hyperparameters (we will use the defaults), then we will use repeated stratified k-fold cross-validation to evaluate the model. We will use three repeats of 10-fold cross-validation, meaning that 10-fold cross-validation is applied three times fitting and evaluating 30 models on the dataset.

The dataset is stratified, meaning that each fold of the cross-validation split will have the same class distribution as the original dataset, in this case, a 1:100 ratio. We will evaluate the model using the ROC area under curve (AUC) metric. This can be optimistic for severely imbalanced datasets but will still show a relative change with better performing models.

```
...
# define model
model = DecisionTreeClassifier()
# evaluate pipeline
cv = RepeatedStratifiedKFold(n_splits=10, n_repeats=3, random_state=1)
scores = cross_val_score(model, X, y, scoring='roc_auc', cv=cv, n_jobs=-1)
```

Once fit, we can calculate and report the mean of the scores across the folds and repeats.

```
...
print('Mean ROC AUC: %.3f' % mean(scores))
```

We would not expect a decision tree fit on the raw imbalanced dataset to perform very well.

Tying this together, the complete example is listed below.

```
# decision tree evaluated on imbalanced dataset
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
# define dataset
X, y = make_classification(n_samples=10000, n_features=2, n_redundant=0,
 n_clusters_per_class=1, weights=[0.99], flip_y=0, random_state=1)
# define model
model = DecisionTreeClassifier()
# evaluate pipeline
cv = RepeatedStratifiedKFold(n_splits=10, n_repeats=3, random_state=1)
scores = cross_val_score(model, X, y, scoring='roc_auc', cv=cv, n_jobs=-1)
print('Mean ROC AUC: %.3f' % mean(scores))
```

Running the example evaluates the model and reports the mean ROC AUC.

Note: Your results may vary 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.

In this case, we can see that a ROC AUC of about 0.76 is reported.

```
Mean ROC AUC: 0.761
```

Now, we can try the same model and the same evaluation method, although use a SMOTE transformed version of the dataset.

The correct application of oversampling during k-fold cross-validation is to apply the method to the training dataset only, then evaluate the model on the stratified but non-transformed test set.

This can be achieved by defining a Pipeline that first transforms the training dataset with SMOTE then fits the model.

```
...
# define pipeline
steps = [('over', SMOTE()), ('model', DecisionTreeClassifier())]
pipeline = Pipeline(steps=steps)
```

This pipeline can then be evaluated using repeated k-fold cross-validation.

Tying this together, the complete example of evaluating a decision tree with SMOTE oversampling on the training dataset is listed below.

```
# decision tree evaluated on imbalanced dataset with SMOTE 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 SMOTE
# define dataset
X, y = make_classification(n_samples=10000, n_features=2, n_redundant=0,
 n_clusters_per_class=1, weights=[0.99], flip_y=0, random_state=1)
# define pipeline
steps = [('over', SMOTE()), ('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='roc_auc', cv=cv, n_jobs=-1)
print('Mean ROC AUC: %.3f' % mean(scores))
```

Running the example evaluates the model and reports the mean ROC AUC score across the multiple folds and repeats.

Note: Your results may vary 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.

In this case, we can see a modest improvement in performance from a ROC AUC of about 0.76 to about 0.80.

```
Mean ROC AUC: 0.809
```

As mentioned in the paper, it is believed that SMOTE performs better when combined with undersampling of the majority class, such as random undersampling.

We can achieve this by simply adding a *RandomUnderSampler* step to the Pipeline.

As in the previous section, we will first oversample the minority class with SMOTE to about a 1:10 ratio, then undersample the majority class to achieve about a 1:2 ratio.

```
...
```

```
# define pipeline
model = DecisionTreeClassifier()
over = SMOTE(sampling_strategy=0.1)
under = RandomUnderSampler(sampling_strategy=0.5)
steps = [('over', over), ('under', under), ('model', model)]
pipeline = Pipeline(steps=steps)
```

Tying this together, the complete example is listed below.

```
# decision tree  on imbalanced dataset with SMOTE oversampling and 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.over_sampling import SMOTE
from imblearn.under_sampling import RandomUnderSampler
# define dataset
X, y = make_classification(n_samples=10000, n_features=2, n_redundant=0,
 n_clusters_per_class=1, weights=[0.99], flip_y=0, random_state=1)
# define pipeline
model = DecisionTreeClassifier()
over = SMOTE(sampling_strategy=0.1)
under = RandomUnderSampler(sampling_strategy=0.5)
steps = [('over', over), ('under', under), ('model', model)]
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='roc_auc', cv=cv, n_jobs=-1)
print('Mean ROC AUC: %.3f' % mean(scores))
```

Running the example evaluates the model with the pipeline of SMOTE oversampling and random undersampling on the training dataset.

Note: Your results may vary 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.

In this case, we can see that the reported ROC AUC shows an additional lift to about 0.83.

```
Mean ROC AUC: 0.834
```

You could explore testing different ratios of the minority class and majority class (e.g. changing the *sampling_strategy* argument) to see if a further lift in performance is possible.

Another area to explore would be to test different values of the k-nearest neighbors selected in the SMOTE procedure when each new synthetic example is created. The default is *k*=*5*, although larger or smaller values will influence the types of examples created, and in turn, may impact the performance of the model.

For example, we could grid search a range of values of *k*, such as values from 1 to 7, and evaluate the pipeline for each value.

```
...
# values to evaluate
k_values = [1, 2, 3, 4, 5, 6, 7]
for k in k_values:
 # define pipeline
 ...
```

The complete example is listed below.

```
# grid search k value for SMOTE oversampling for imbalanced classification
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 SMOTE
from imblearn.under_sampling import RandomUnderSampler
# define dataset
X, y = make_classification(n_samples=10000, n_features=2, n_redundant=0,
 n_clusters_per_class=1, weights=[0.99], flip_y=0, random_state=1)
# values to evaluate
k_values = [1, 2, 3, 4, 5, 6, 7]
for k in k_values:
 # define pipeline
 model = DecisionTreeClassifier()
 over = SMOTE(sampling_strategy=0.1, k_neighbors=k)
 under = RandomUnderSampler(sampling_strategy=0.5)
 steps = [('over', over), ('under', under), ('model', model)]
 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='roc_auc', cv=cv, n_jobs=-1)
 score = mean(scores)
 print('> k=%d, Mean ROC AUC: %.3f' % (k, score))
```

Running the example will perform SMOTE oversampling with different k values for the KNN used in the procedure, followed by random undersampling and fitting a decision tree on the resulting training dataset.

The mean ROC AUC is reported for each configuration.

Note: Your results may vary 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.

In this case, the results suggest that a *k=3* might be good with a ROC AUC of about 0.84, and *k=7* might also be good with a ROC AUC of about 0.85.

This highlights that both the amount of oversampling and undersampling performed (sampling_strategy argument) and the number of examples selected from which a partner is chosen to create a synthetic example (*k_neighbors*) may be important parameters to select and tune for your dataset.

```
> k=1, Mean ROC AUC: 0.827
> k=2, Mean ROC AUC: 0.823
> k=3, Mean ROC AUC: 0.834
> k=4, Mean ROC AUC: 0.840
> k=5, Mean ROC AUC: 0.839
> k=6, Mean ROC AUC: 0.839
> k=7, Mean ROC AUC: 0.853
```

Now that we are familiar with how to use SMOTE when fitting and evaluating classification models, let's look at some extensions of the SMOTE procedure.

SMOTE With Selective Synthetic Sample Generation

We can be selective about the examples in the minority class that are oversampled using SMOTE.

In this section, we will review some extensions to SMOTE that are more selective regarding the examples from the minority class that provide the basis for generating new synthetic examples.

Borderline-SMOTE

A popular extension to SMOTE involves selecting those instances of the minority class that are misclassified, such as with a k-nearest neighbor classification model.

We can then oversample just those difficult instances, providing more resolution only where it may be required.

> *The examples on the borderline and the ones nearby [...] are more apt to be misclassified than the ones far from the borderline, and thus more important for classification.*

— Borderline-SMOTE: A New Over-Sampling Method in Imbalanced Data Sets Learning, 2005.

These examples that are misclassified are likely ambiguous and in a region of the edge or border of decision boundary where class membership may overlap. As such, this modified to SMOTE is called Borderline-SMOTE and was proposed by Hui Han, et al. in their 2005 paper titled "Borderline-SMOTE: A New Over-Sampling Method in Imbalanced Data Sets Learning."

The authors also describe a version of the method that also oversampled the majority class for those examples that cause a misclassification of borderline instances in the minority class. This is referred to as Borderline-SMOTE1, whereas the oversampling of just the borderline cases in minority class is referred to as Borderline-SMOTE2.

> *Borderline-SMOTE2 not only generates synthetic examples from each example in DANGER and its positive nearest neighbors in P, but also does that from its nearest negative neighbor in N.*

— Borderline-SMOTE: A New Over-Sampling Method in Imbalanced Data Sets Learning, 2005.

We can implement Borderline-SMOTE1 using the BorderlineSMOTE class from imbalanced-learn.

We can demonstrate the technique on the synthetic binary classification problem used in the previous sections.

Instead of generating new synthetic examples for the minority class blindly, we would expect the Borderline-SMOTE method to only create synthetic examples along the decision boundary between the two classes.

The complete example of using Borderline-SMOTE to oversample binary classification datasets is listed below.

```
# borderline-SMOTE for imbalanced dataset
from collections import Counter
from sklearn.datasets import make_classification
```

```
from imblearn.over_sampling import BorderlineSMOTE
from matplotlib import pyplot
from numpy import where
# define dataset
X, y = make_classification(n_samples=10000, n_features=2, n_redundant=0,
 n_clusters_per_class=1, weights=[0.99], flip_y=0, random_state=1)
# summarize class distribution
counter = Counter(y)
print(counter)
# transform the dataset
oversample = BorderlineSMOTE()
X, y = oversample.fit_resample(X, y)
# summarize the new class distribution
counter = Counter(y)
print(counter)
# scatter plot of examples by class label
for label, _ in counter.items():
 row_ix = where(y == label)[0]
 pyplot.scatter(X[row_ix, 0], X[row_ix, 1], label=str(label))
pyplot.legend()
pyplot.show()
```

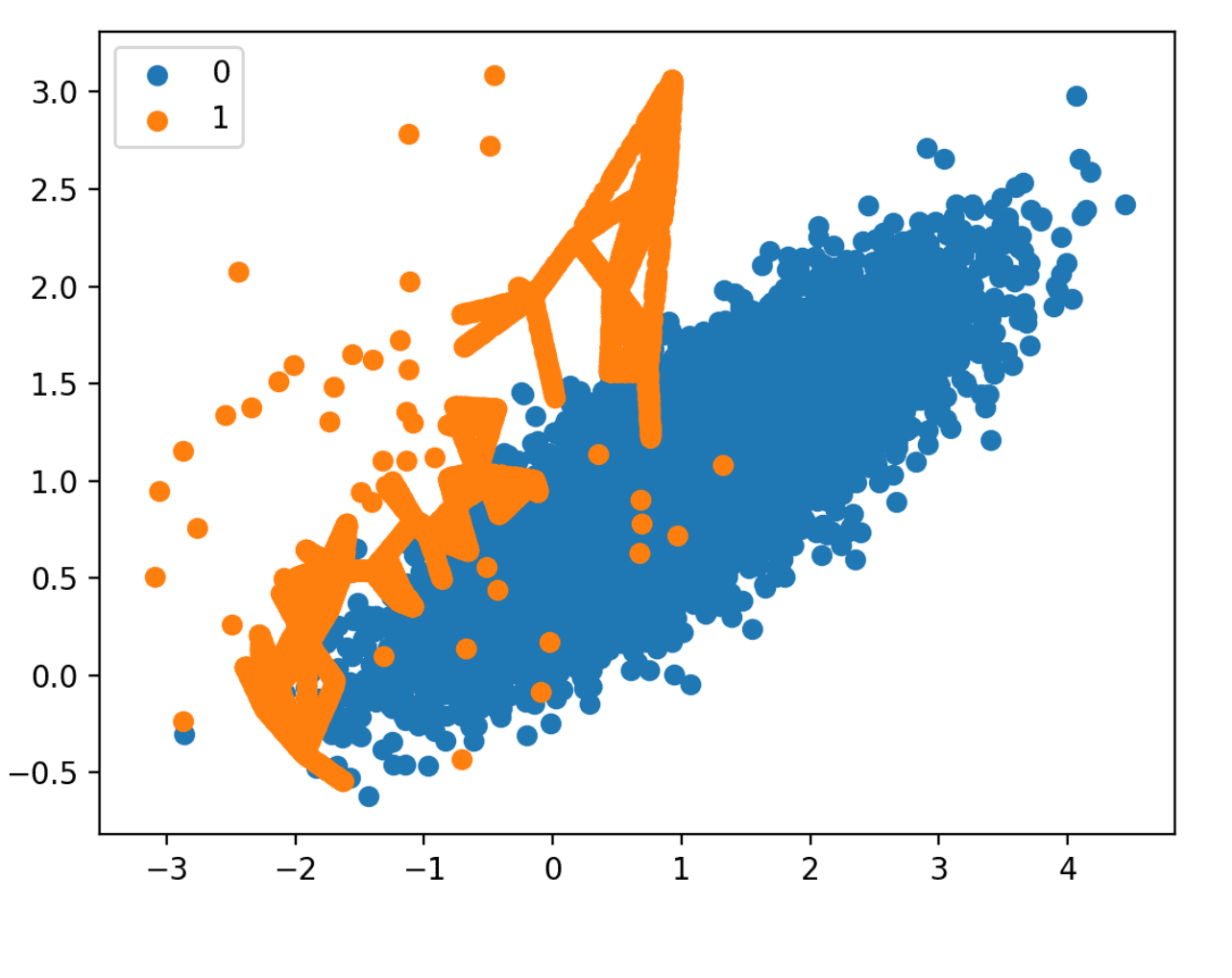
Running the example first creates the dataset and summarizes the initial class distribution, showing a 1:100 relationship.

The Borderline-SMOTE is applied to balance the class distribution, which is confirmed with the printed class summary.

```
Counter({0: 9900, 1: 100})
Counter({0: 9900, 1: 9900})
```

Finally, a scatter plot of the transformed dataset is created. The plot clearly shows the effect of the selective approach to oversampling. Examples along the decision boundary of the minority class are oversampled intently (orange).

The plot shows that those examples far from the decision boundary are not oversampled. This includes both examples that are easier to classify (those orange points toward the top left of the plot) and those that are overwhelmingly difficult to classify given the strong class overlap (those orange points toward the bottom right of the plot).



Scatter Plot of Imbalanced Dataset With Borderline-SMOTE Oversampling

Borderline-SMOTE SVM

Hien Nguyen, et al. suggest using an alternative of Borderline-SMOTE where an SVM algorithm is used instead of a KNN to identify misclassified examples on the decision boundary.

Their approach is summarized in the 2009 paper titled "Borderline Over-sampling For Imbalanced Data Classification." An SVM is used to locate the decision boundary defined by the support vectors and examples in the minority class that close to the support vectors become the focus for generating synthetic examples.

> *... the borderline area is approximated by the support vectors obtained after training a standard SVMs classifier on the original training set. New instances will be randomly created along the lines joining each minority class support vector with a number of its nearest neighbors using the interpolation*

— Borderline Over-sampling For Imbalanced Data Classification, 2009.

In addition to using an SVM, the technique attempts to select regions where there are fewer examples of the minority class and tries to extrapolate towards the class boundary.

> *If majority class instances count for less than a half of its nearest neighbors, new instances will be created with extrapolation to expand minority class area toward the majority class.*

— Borderline Over-sampling For Imbalanced Data Classification, 2009.

This variation can be implemented via the SVMSMOTE class from the imbalanced-learn library.

The example below demonstrates this alternative approach to Borderline SMOTE on the same imbalanced dataset.

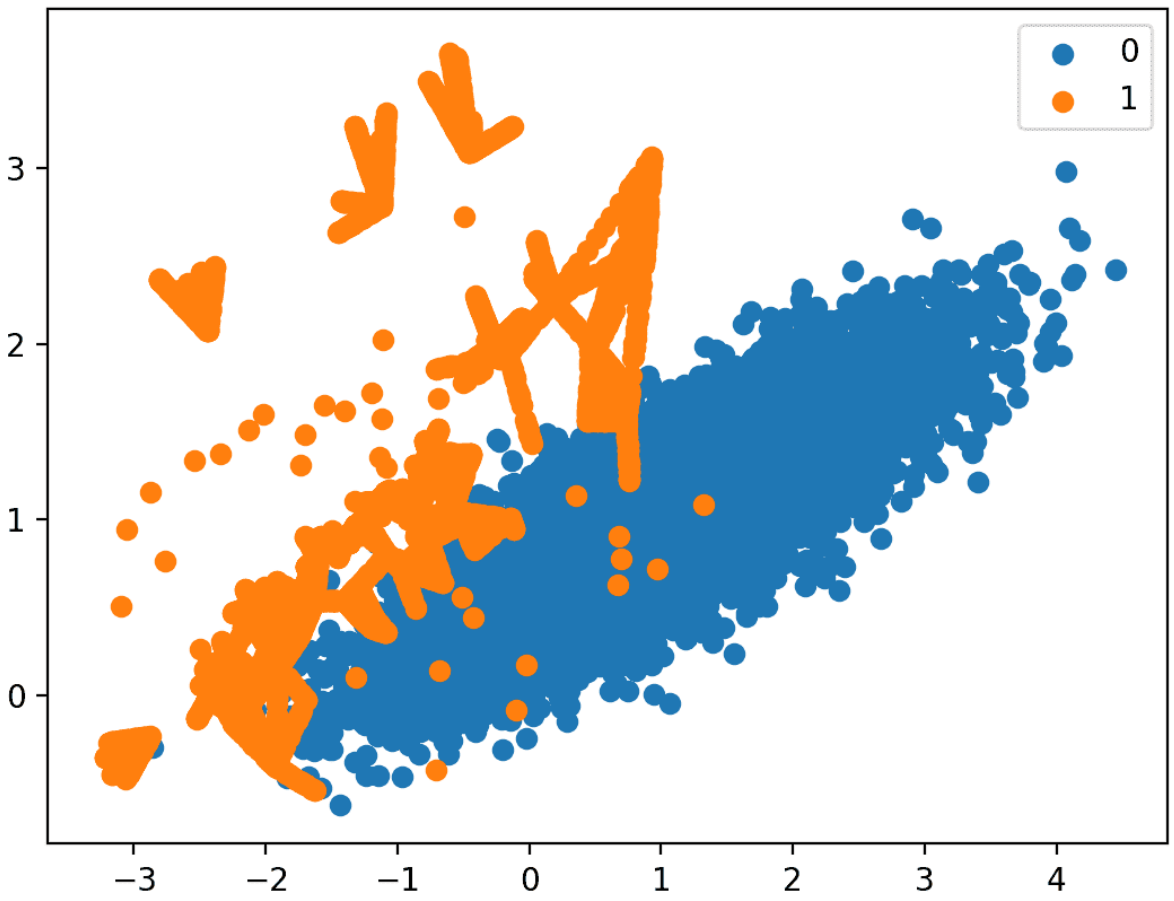
```
# borderline-SMOTE with SVM for imbalanced dataset
from collections import Counter
from sklearn.datasets import make_classification
from imblearn.over_sampling import SVMSMOTE
from matplotlib import pyplot
from numpy import where
# define dataset
X, y = make_classification(n_samples=10000, n_features=2, n_redundant=0,
 n_clusters_per_class=1, weights=[0.99], flip_y=0, random_state=1)
# summarize class distribution
counter = Counter(y)
print(counter)
# transform the dataset
oversample = SVMSMOTE()
X, y = oversample.fit_resample(X, y)
# summarize the new class distribution
counter = Counter(y)
print(counter)
# scatter plot of examples by class label
for label, _ in counter.items():
 row_ix = where(y == label)[0]
 pyplot.scatter(X[row_ix, 0], X[row_ix, 1], label=str(label))
pyplot.legend()
pyplot.show()
```

Running the example first summarizes the raw class distribution, then the balanced class distribution after applying Borderline-SMOTE with an SVM model.

```
Counter({0: 9900, 1: 100})
Counter({0: 9900, 1: 9900})
```

A scatter plot of the dataset is created showing the directed oversampling along the decision boundary with the majority class.

We can also see that unlike Borderline-SMOTE, more examples are synthesized away from the region of class overlap, such as toward the top left of the plot.



Scatter Plot of Imbalanced Dataset With Borderline-SMOTE Oversampling With SVM

Adaptive Synthetic Sampling (ADASYN)

Another approach involves generating synthetic samples inversely proportional to the density of the examples in the minority class.

That is, generate more synthetic examples in regions of the feature space where the density of minority examples is low, and fewer or none where the density is high.

This modification to SMOTE is referred to as the Adaptive Synthetic Sampling Method, or ADASYN, and was proposed to Haibo He, et al. in their 2008 paper named for the method titled "ADASYN: Adaptive Synthetic Sampling Approach For Imbalanced Learning."

> *ADASYN is based on the idea of adaptively generating minority data samples according to their distributions: more synthetic data is generated for minority class samples that are harder to learn compared to those minority samples that are easier to learn.*

— ADASYN: Adaptive synthetic sampling approach for imbalanced learning, 2008.

With online Borderline-SMOTE, a discriminative model is not created. Instead, examples in the minority class are weighted according to their density, then those examples with the lowest density are the focus for the SMOTE synthetic example generation process.

> *The key idea of ADASYN algorithm is to use a density distribution as a criterion to automatically decide the number of synthetic samples that need to be generated for each minority data example.*

— ADASYN: Adaptive synthetic sampling approach for imbalanced learning, 2008.

We can implement this procedure using the ADASYN class in the imbalanced-learn library.

The example below demonstrates this alternative approach to oversampling on the imbalanced binary classification dataset.

```
# Oversample and plot imbalanced dataset with ADASYN
from collections import Counter
from sklearn.datasets import make_classification
from imblearn.over_sampling import ADASYN
from matplotlib import pyplot
from numpy import where
# define dataset
X, y = make_classification(n_samples=10000, n_features=2, n_redundant=0,
 n_clusters_per_class=1, weights=[0.99], flip_y=0, random_state=1)
# summarize class distribution
counter = Counter(y)
print(counter)
# transform the dataset
oversample = ADASYN()
X, y = oversample.fit_resample(X, y)
# summarize the new class distribution
counter = Counter(y)
print(counter)
# scatter plot of examples by class label
for label, _ in counter.items():
 row_ix = where(y == label)[0]
 pyplot.scatter(X[row_ix, 0], X[row_ix, 1], label=str(label))
pyplot.legend()
pyplot.show()
```

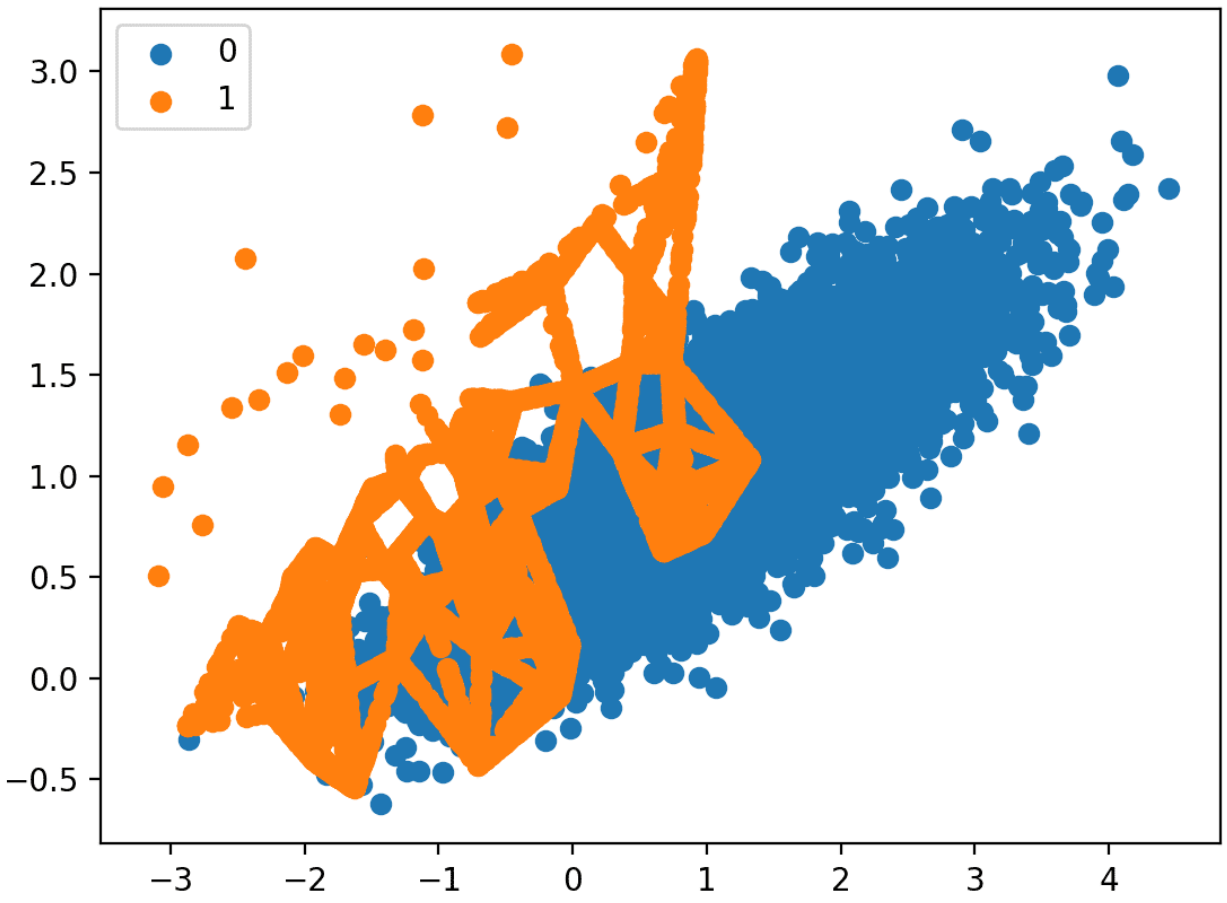
Running the example first creates the dataset and summarizes the initial class distribution, then the updated class distribution after oversampling was performed.

```
Counter({0: 9900, 1: 100})
Counter({0: 9900, 1: 9899})
```

A scatter plot of the transformed dataset is created. Like Borderline-SMOTE, we can see that synthetic sample generation is focused around the decision boundary as this region has the lowest density.

Unlike Borderline-SMOTE, we can see that the examples that have the most class overlap have the most focus. On problems where these low density examples might be outliers, the ADASYN approach may put too much attention on these areas of the feature space, which may result in worse model performance.

It may help to remove outliers prior to applying the oversampling procedure, and this might be a helpful heuristic to use more generally.



Scatter Plot of Imbalanced Dataset With Adaptive Synthetic Sampling (ADASYN)

Further Reading

This section provides more resources on the topic if you are looking to go deeper.

Books

- Learning from Imbalanced Data Sets, 2018.
- Imbalanced Learning: Foundations, Algorithms, and Applications, 2013.

Papers

- SMOTE: Synthetic Minority Over-sampling Technique, 2002.
- Borderline-SMOTE: A New Over-Sampling Method in Imbalanced Data Sets Learning, 2005.
- Borderline Over-sampling For Imbalanced Data Classification, 2009.
- ADASYN: Adaptive Synthetic Sampling Approach For Imbalanced Learning, 2008.

API

- imblearn.over_sampling.SMOTE API.
- imblearn.over_sampling.SMOTENC API.
- imblearn.over_sampling.BorderlineSMOTE API.
- imblearn.over_sampling.SVMSMOTE API.
- imblearn.over_sampling.ADASYN API.

Articles

- Oversampling and undersampling in data analysis, Wikipedia.

Summary

In this tutorial, you discovered the SMOTE for oversampling imbalanced classification datasets.

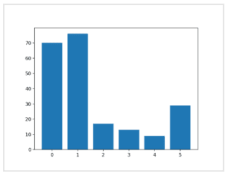
Specifically, you learned:

- How the SMOTE synthesizes new examples for the minority class.
- How to correctly fit and evaluate machine learning models on SMOTE-transformed training datasets.
- How to use extensions of the SMOTE that generate synthetic examples along the class decision boundary.

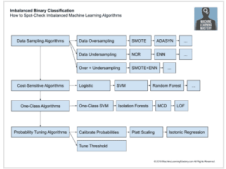
Do you have any questions?
Ask your questions in the comments below and I will do my best to answer.

More On This Topic

How to Combine Oversampling and Undersampling for…



Multi-Class Imbalanced Classification



Step-By-Step Framework for Imbalanced Classification…