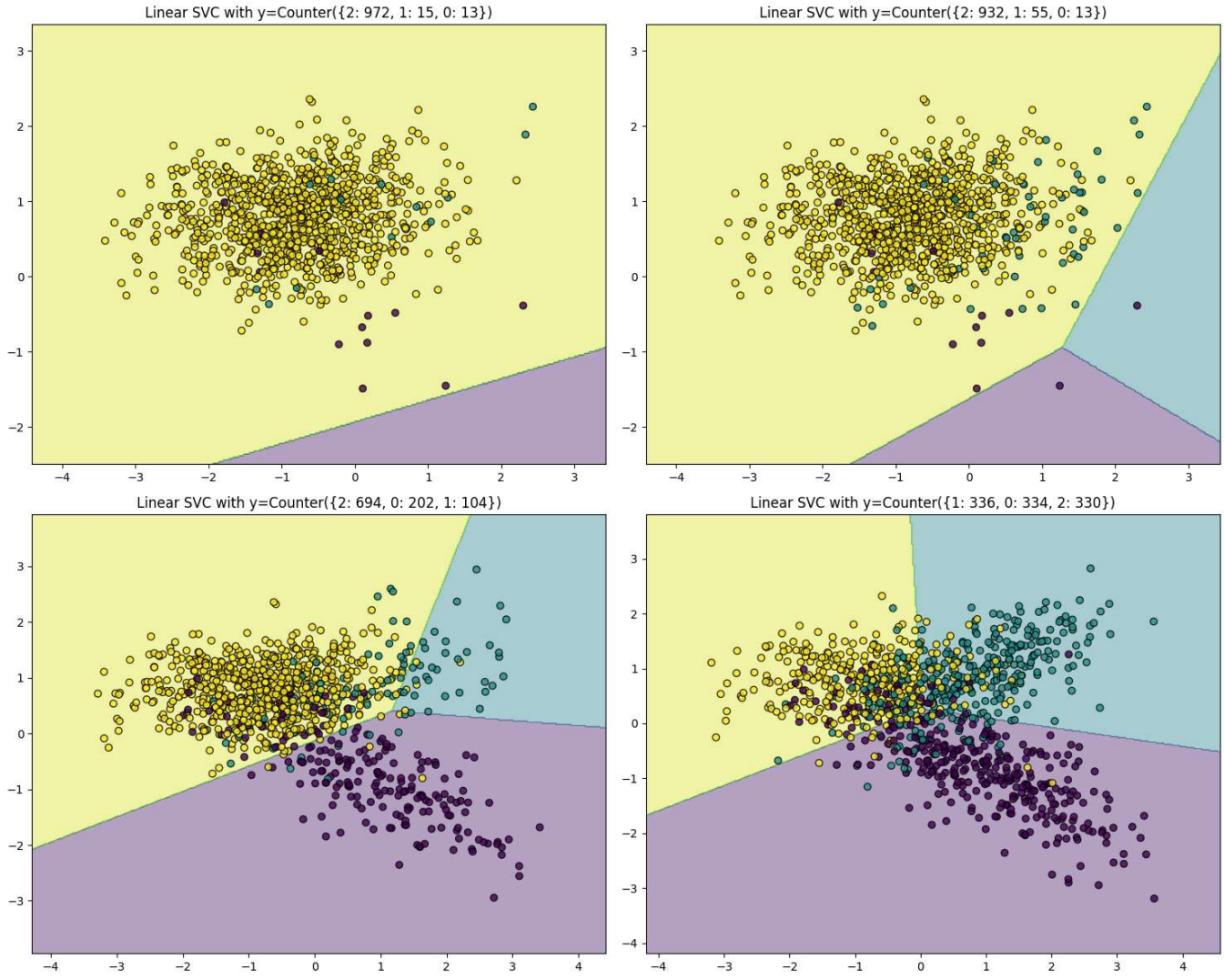
A Deep Dive Into Imbalanced Data: Over-Sampling

Learn how to use imbalanced-learn to improve your performance

Lukas Frei Follow
May 26 · 9 min read ★

https://unsplash.com/photos/nvDJfbFv0pI

When implementing classification algorithms, the structure of your data is of great significance. Specifically, the balance between the number of observations for each potential output heavily influences your prediction's performance (I intentionally avoided using the word "accuracy" for reasons I will later elaborate on in greater detail). Let's illustrate that with a plot from imbalanced-learn's documentation:



As you can see, we are dealing with a data set that has three different classes that we are trying to predict with the help of a linear support vector machine algorithm. In the top-left plot, the yellow class is clearly dominating the other classes and the decision boundaries for the other classes are barely recognizable. As we keep adding more observations to the underrepresented classes, the decision boundaries change dramatically. Thus, when trying to make predictions with regards to a minority class, we need to find ways to avoid being misled by the majority class.

Installing imblearn

Before getting started with imbalanced data we have to install imblearn. Generally speaking, there are two ways to install packages in Python: via **pip install** and via **conda install**. Alternatively, you could also clone the GitHub repository.

```
# installing imblearn using pip
pip install -U imbalanced-learn
```

```
pip install -U imbalanced-learn

# installing imblearn using conda
conda install -c conda-forge imbalanced-learn
```

installing_imblearn.py hosted with ♡ by GitHub view raw

The Data

After having installed imblearn and its dependencies we can get started working on imbalanced data. For the purpose of this article, I am going to use the Credit Fraud data set that I worked on in a previous article. In case you are not familiar with the data, let's do some quick EDA to gain a rudimentary understanding of what the data looks like and why we need to utilize special techniques to achieve acceptable results.

```
# importing packages
import numpy as np
import pandas as pd
import scipy.stats as stats
import sklearn
import keras
import imblearn
import matplotlib.pyplot as plt
import seaborn as sns
plt.style.use('ggplot')

# reading in CSV
df = pd.read_csv('creditcard.csv')
df.sample(5)
```

eda_imblearn.py hosted with ♡ by GitHub view raw

The code above will result in the following output:

V5	V6	V7	V8	V9	...	V21	V22	V23	V24	V25	V26	V27	V28	Amount	Class
-1.645794	-1.043947	-0.972180	-0.191960	-1.688581	...	-0.222866	-0.313256	0.041431	0.341415	0.350738	-0.203650	0.021897	0.019911	20.00	0
0.603019	-1.392093	-0.330570	0.417282	-0.480147	...	0.228879	0.233850	-0.115103	0.510081	-0.292391	1.155064	-0.048749	-0.164954	2.37	0
-0.515124	0.354954	0.881697	0.442993	-0.464449	...	-0.320608	-1.209128	0.331781	-0.076616	0.448085	-0.807567	-0.065077	-0.005213	203.90	0
-1.033677	1.146805	-0.235072	1.135943	-0.437181	...	0.197768	0.601002	-0.173496	-0.266810	0.051877	-0.140848	-0.105420	-0.100677	132.72	0
-1.011201	0.159212	-0.508960	1.094697	-0.318561	...	0.070957	0.512048	-0.090333	0.588702	0.105039	-0.205072	0.307944	0.111419	4.90	0

V1 through V28 have been anonymized using a nondisclosed dimensionality reduction technique. The only named variables are the time of the transaction, the amount, as well a variable indicating whether or not the transaction was fraudulent (0: not fraudulent, 1: fraudulent). Since most transactions are not fraudulent we are dealing with a heavily imbalanced data set. Let us visualize the imbalance to understand the need for using imblearn:

```
# taking a closer look at the class variable
sns.countplot('Class', data = df)
plt.title('No Fraud (0) vs. Fraud (1)')
```

class_comparison.py hosted with ♡ by GitHub view raw

The code chunk above will yield the following bar chart:

As expected, the vast majority of transactions is not fraudulent. Thus, building a reliable classifier to detect fraudulent transactions using the same techniques as one might use on a balanced data set will most certainly lead to a very bad performance.

Over-sampling

One common way to tackle the issue of imbalanced data is over-sampling. Over-sampling refers to various methods that aim to increase the number of instances from the underrepresented class in the data set. In our case, these techniques will increase the number of fraudulent transactions in our data (usually to 50:50). You might ask why one would even do that in the first place. Good question. If you do not balance the number of instances, most classification algorithms will heavily focus on the majority class. As a result, it might seem like your algorithm is achieving superb results when, in reality, it is simply always predicting the majority class.

Random naive over-sampling

The easiest way to do so is to randomly select observations from the minority class and add them to the data set until we achieve a balance between the majority and minority class. Since this is relatively straight-forward one would not necessarily have to use imblearn but could randomly select observations using NumPy for instance.

In imblearn, one would do the following after splitting our data into x (all variables except for the class) and y (the class):

```
# imports
from imblearn.over_sampling import RandomOverSampler

# random oversampling
ros = RandomOverSampler(random_state=0)
X_resampled, y_resampled = ros.fit_resample(x, y)

# using Counter to display results of naive oversampling
from collections import Counter
print(sorted(Counter(y_resampled).items()))
```

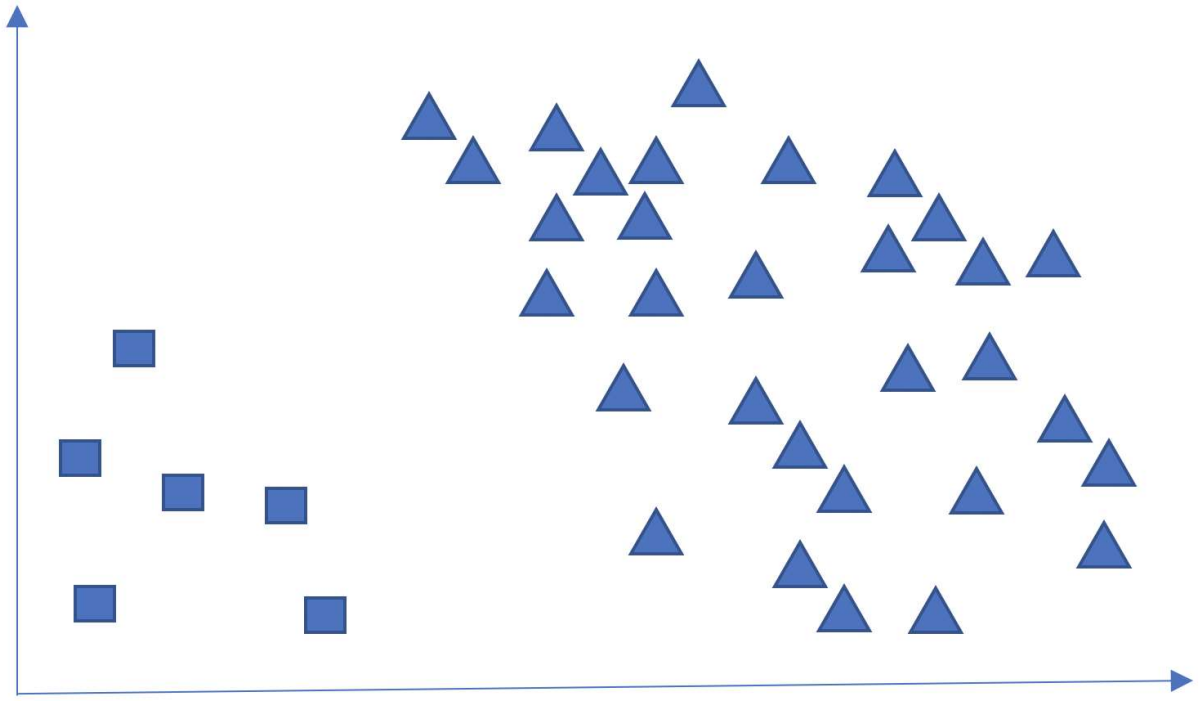
random_naive_oversampling.py hosted with ♡ by GitHub view raw

The result of this code chunk with an equal amount of fraudulent and non-fraudulent transactions. To be more specific, y would have 284,315 fraudulent and 284,315 non-fraudulent observations. Now, instead of using the original, imbalanced data, one would use this augmented data set to train a classification algorithm.

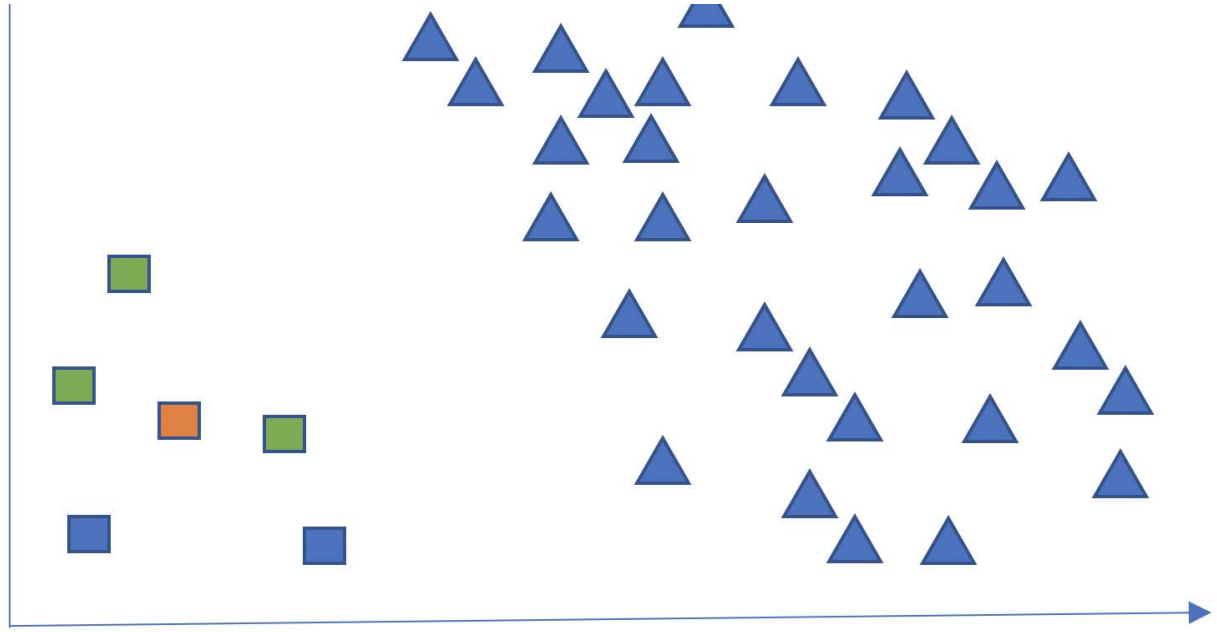
One issue with random naive over-sampling is that it just duplicates already existing data. Therefore, while classification algorithms are exposed to a greater amount of observations from the minority class, they won't learn more about how to tell fraudulent and non-fraudulent observations apart. The new data does not contain more information about the characteristics of fraudulent transactions than the old data.

Synthetic Minority Over-Sampling Technique (SMOTE)

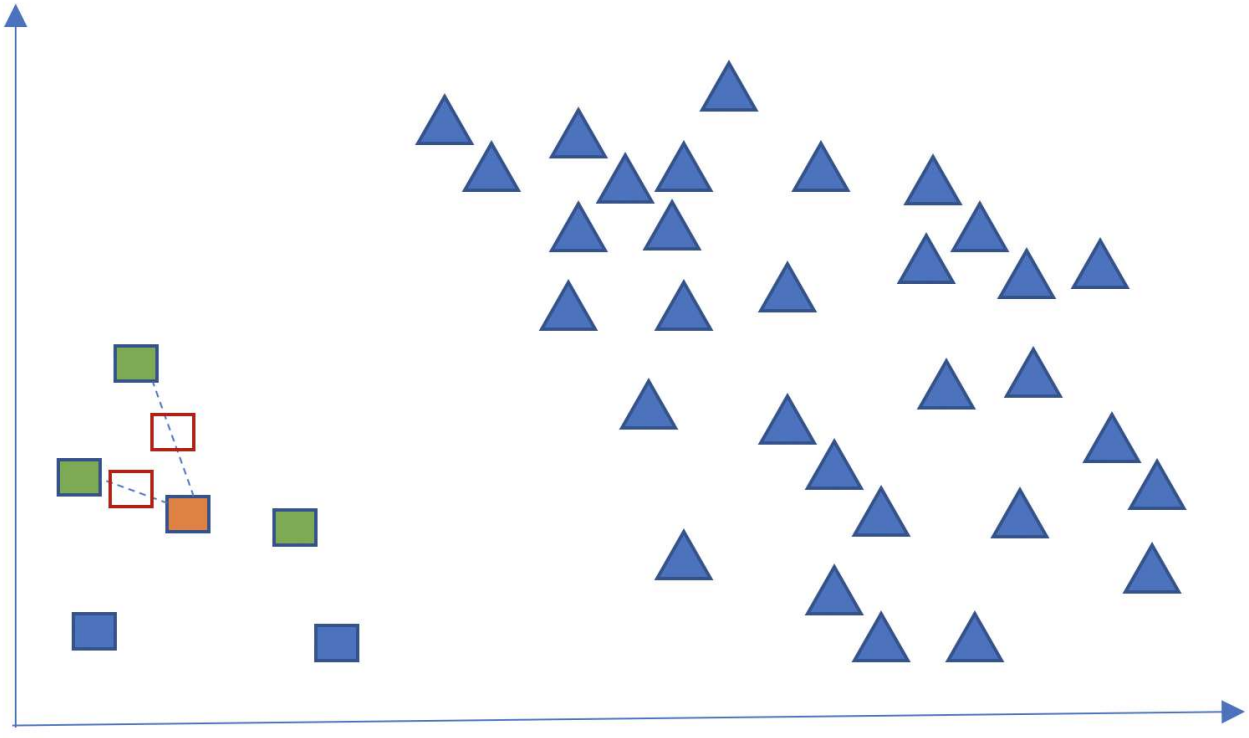
A more advanced alternative to using random naive over-sampling is Synthetic Minority Over-Sampling Technique(SMOTE). While SMOTE still oversamples the minority class, it does not rely on reusing previously existing observations. Instead, SMOTE creates new (synthetic) observations based on the observations in your data. How does SMOTE do that? To illustrate my point, I've put together a fictional data set:



As you can see, there are way more triangles than squares in this fictional data set. Now, in order to train a more accurate classifier, we would like to utilize SMOTE to oversample the squares. First, SMOTE finds the k-nearest-neighbors of each member of the minority class. Let's visualize that for one of the squares and assume that k equals three:



In the visualization above we have identified the three nearest neighbors of the orange square. Now, depending on how much oversampling is desired, one or more of these nearest neighbors are going to be used to create new observations. For the purpose of this explanation, let us assume that we are going to use two of the three nearest neighbors to create new observations. The next and final step is to create new observations by randomly choosing a point on the line connecting the observation with its nearest neighbor:



The dashed lines represent the connection between the orange square and its green nearest neighbors. The two red squares denote the new observations added to the data set by SMOTE.

SMOTE's main advantage compared to traditional random naive over-sampling is that by creating synthetic observations instead of reusing existing observations, your classifier is less likely to overfit. At the same time, you should always make sure that the observations created by SMOTE are realistic. The technique SMOTE uses to create new observations only helps if the synthetic observations are realistic and could have been observed in reality.

After going through the theory, it is time to implement SMOTE in imblearn:

```
# importing SMOTE
from imblearn.over_sampling import SMOTE

# applying SMOTE to our data and checking the class counts
X_resampled, y_resampled = SMOTE().fit_resample(x, y)
print(sorted(Counter(y_resampled).items()))
```

SMOTE.py hosted with ♡ by GitHub view raw

Running the code chunk above will output a class count that proves that there is an equal amount of observations in both classes.

Adaptive Synthetic (ADASYN)

Adaptive Synthetic (ADASYN) provides an alternative to using SMOTE. Let us walk through the algorithm step by step.

First, ADASYN calculates the ratio of minority to majority observations

$$d = \frac{m_s}{m_l}$$

Next, ADASYN computes the total number of synthetic minority data to generate:

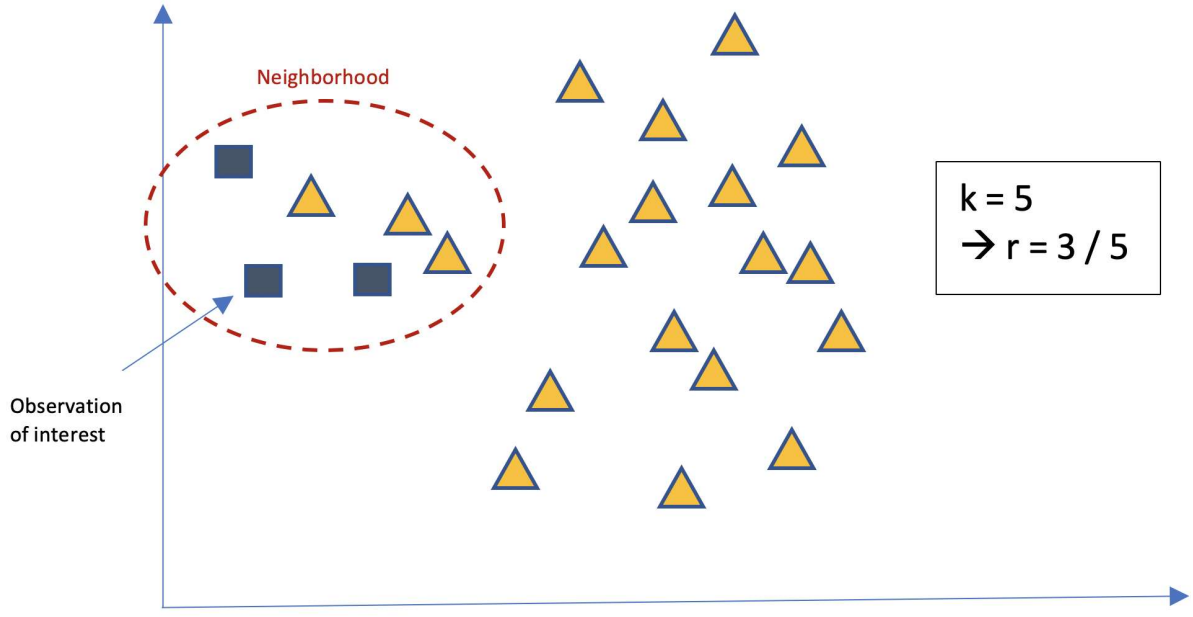
$$G = (m_l - m_s)\beta$$

Here, G is the total number of synthetic minority data to generate and ß denotes the ratio of minority to majority observations. Thus, ß = 1 would mean that there are equally as many observations in both classes after using ADASYN.

Third, ADASYN finds the k-nearest neighbors for each of the minority observations and computes an r value:

$$r_i = \frac{\#majority}{k}$$

The r_i value measures the dominance of the majority class in the neighborhood. The higher r_i, the more dominant the majority class and the more difficult the neighborhood is to learn for your classifier. Let us calculate r_i for some fictional minority observation:



Three of the highlighted minority observation's five nearest neighbors belong to the majority class. Thus, r for this observation is equal to 3/5.

Before actually computing the number of synthetic observations that are going to be created, ADASYN normalizes the r values for all minority observations so that their sum is one.

$$\hat{r}_i = \frac{r_i}{\sum r_i}$$

$$\sum \hat{r}_i = 1$$

Next, ADASYN computes the number of synthetic observations to generate in each neighborhood:

$$G_i = G\hat{r}_i$$

Since G_i is calculated using the respective r_i value, ADASYN will create more synthetic observations in neighborhoods with a greater ratio of majority to minority observations. Therefore, the classifier will have more observations to learn from in these difficult areas.

Finally, ADASYN generates synthetic observations:

$$s_i = x_i + (x_{zi} - x_i)\lambda$$

One could do so using a linear combination of the observation of interest and a neighbor similar to SMOTE or utilize more advanced techniques like drawing a plane between three minority observations and randomly selecting a point on that plane.

ADASYN's main advantage lies in its adaptive nature: by basing the number of synthetic observations on the ratio of majority to minority observations, ADASYN places a higher emphasis on more challenging regions of the data.

As with SMOTE, after discussing the theory, it is time to look at the code:

```python
# importing ADASYN
from imblearn.over_sampling import ADASYN

# applying ADASYN
X_resampled, y_resampled = ADASYN().fit_resample(x, y)
print(sorted(Counter(y_resampled).items()))
```

ADASYN.py hosted with ♡ by GitHub view raw

As you can see, imblearn's syntax is easy to memorize and resembles sklearn's syntax. Thus, if you are used to sklearn's syntax, you won't have to put in a lot of effort to include imblearn in your machine learning workflow.

SMOTE Extensions

As with most algorithms, there are several extensions of SMOTE. These aim to improve SMOTE by adding to its functionality or lessening its weaknesses. Examples of SMOTE extensions that can be found in imblearn include:

- **BorderlineSMOTE:** Instead of oversampling between all minority observations, BorderlineSMOTE aims to increase the number of minority observations that border majority observations. The goal here is to allow the classifier to be able to distinguish between these borderline observations more clearly.
- **SVMSMOTE:** SVMSMOTE, as its name implies, utilizes the Support Vector Machine algorithm to generate new minority observations close to the border between the majority and minority classes.

Here is exemplary code for BorderlineSMOTE in imblearn:

```python
# BorderlineSMOTE
from imblearn.over_sampling import BorderlineSMOTE

X_resampled, y_resampled = BorderlineSMOTE().fit_resample(x, y)
print(sorted(Counter(y_resampled).items()))
```

SMOTEExtensions.py hosted with ♡ by GitHub view raw

Conclusion

Dealing with imbalanced data can be extremely challenging. However, imblearn provides a neat way to incorporate techniques that battle imbalance into your machine learning workflow in sklearn. By first understanding these techniques and then utilizing them, imbalanced data should become a lot less intimidating. Besides over-sampling, there are several other ways to attack minority, such as under-sampling or combinations of the two. In the next post of this deep-dive, I am going to tackle under-sampling in a similar fashion.

. . .

References:

[1] N. V. Chawla, K. W. Bowyer, L. O.Hall, W. P. Kegelmeyer, "SMOTE: synthetic minority over-sampling technique," Journal of artificial intelligence research, 321–357, 2002.

[2] He, Haibo, Yang Bai, Edwardo A. Garcia, and Shutao Li. "ADASYN: Adaptive synthetic sampling approach for imbalanced learning," In IEEE International Joint Conference on Neural Networks (IEEE World Congress on Computational Intelligence), pp. 1322–1328, 2008.

[3] H. Han, W. Wen-Yuan, M. Bing-Huan, "Borderline-SMOTE: a new over-sampling method in imbalanced data sets learning," Advances in intelligent computing, 878–887, 2005.

[4] H. M. Nguyen, E. W. Cooper, K. Kamei, "Borderline over-sampling for imbalanced data classification," International Journal of Knowledge Engineering and Soft Data Paradigms, 3(1), pp.4–21, 2009.

[5] imbalanced-learn's documentation