Overcoming Class Imbalance using SMOTE Techniques

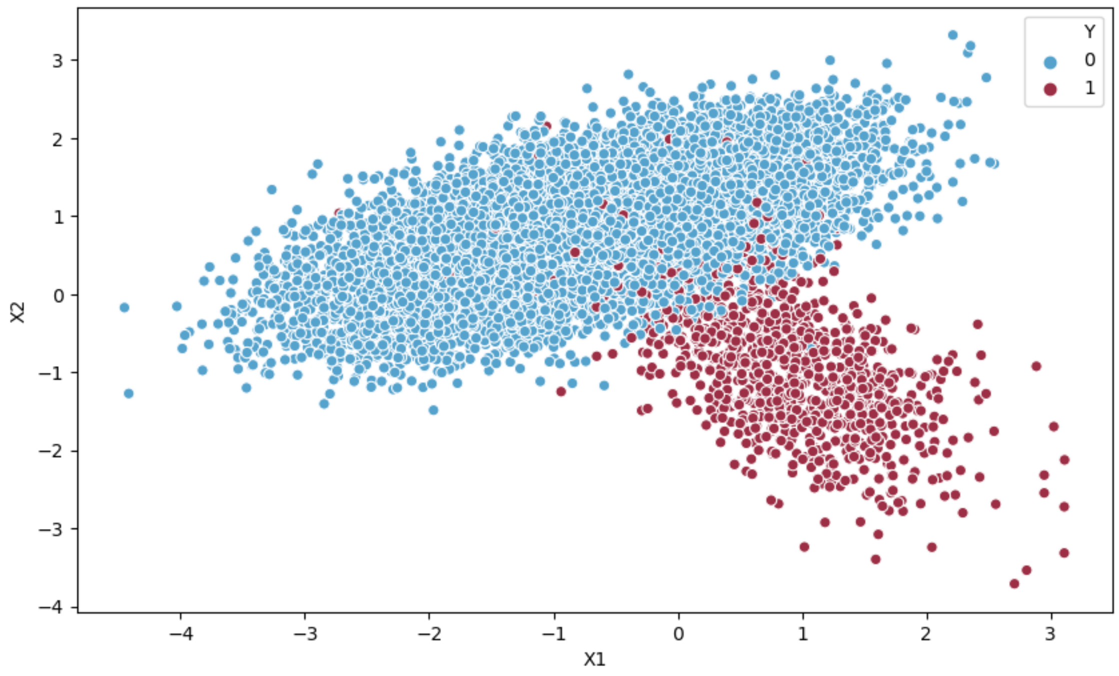
CLASSIFICATION INTERMEDIATE MACHINE LEARNING PYTHON STRUCTURED DATA TECHNIQUE

This article was published as a part of the Data Science Blogathon.

Disclaimer: In this article, I'll cover some resampling techniques to handle imbalanced data. Therefore assuming the readers have some knowledge related to the binary classification problem.

Introduction

Today any machine learning practitioners working with binary classification problems must have come across this typical situation of an imbalanced dataset. This is a typical scenario seen across many valid business problems like fraud detection, spam filtering, rare disease discovery, hardware fault detection, etc. Class imbalance is a scenario that arises when we have unequal distribution of class in a dataset i.e. the no. of data points in negative class (majority class) very large compared to that of the positive class (minority class).



Generally, the minority/positive class is the class of interest and we aim to achieve the best results in this class rather. If the imbalanced data is not treated beforehand, then this will degrade the performance of the classifier model. Most of the predictions will correspond to the majority class and treat the minority class features as noise in the data and ignore them. This will result in a high bias in the model.

The Accuracy Paradox

Suppose, you're working in health insurance based fraud detection problem. In such problems, we generally observe that in every 100 insurance claims 99 of them are non-fraudulent and 1 is fraudulent. So a binary classifier model need not be a complex model to predict all outcomes as 0 meaning non-fraudulent and achieve a great accuracy of 99%. Clearly, in such cases where class distribution is skewed, the accuracy metric is biased and not preferable.

Dealing with Imbalanced Data

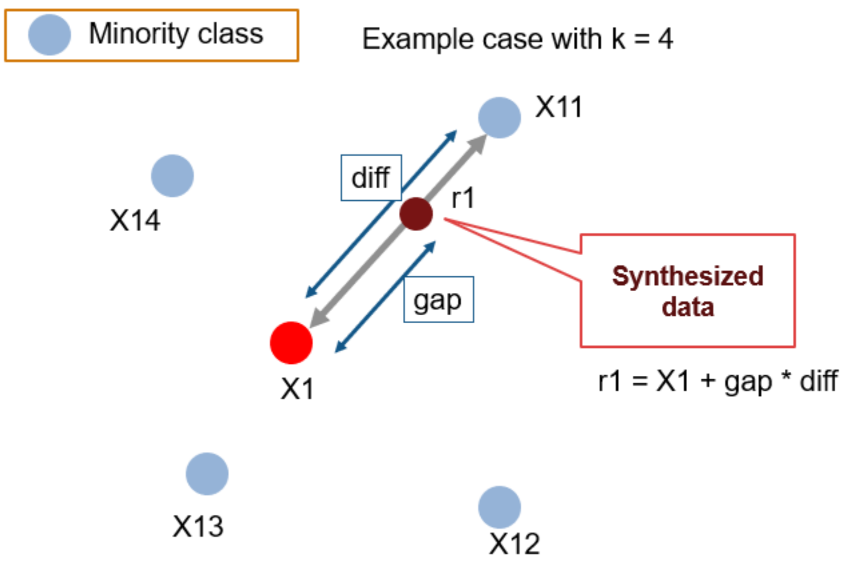
Resampling data is one of the most commonly preferred approaches to deal with an imbalanced dataset. There are broadly two types of methods for this i) Undersampling ii) Oversampling. In most of the cases, oversampling is preferred over undersampling techniques. The reason being, in undersampling we tend to remove instances from data that may be carrying some important information. In this article, I am specifically covering some special data augmentation oversampling techniques: SMOTE and it's related counterparts.

1. SMOTE: Synthetic Minority Oversampling Technique

SMOTE is an oversampling technique where the synthetic samples are generated for the minority class. This algorithm helps to overcome the overfitting problem posed by random oversampling. It focuses on the feature space to generate new instances with the help of interpolation between the positive instances that lie together.

Working Procedure:

At first the total no. of oversampling observations, N is set up. Generally, it is selected such that the binary class distribution is 1:1. But that could be tuned down based on need. Then the iteration starts by first selecting a positive class instance at random. Next, the KNN's (by default 5) for that instance is obtained. At last, N of these K instances is chosen to interpolate new synthetic instances. To do that, using any distance metric the difference in distance between the feature vector and its neighbors is calculated. Now, this difference is multiplied by any random value in (0,1] and is added to the previous feature vector. This is pictorially represented below:



Source: https://github.com/minoue-xx/Oversampling-Imbalanced-Data

Python code for SMOTE algorithm

Though this algorithm is quite useful, it has few drawbacks associated with it.

i) The synthetic instances generated are in the same direction i.e. connected by an artificial line its diagonal instances. This in turn complicates the decision surface generated by few classifier algorithms.

ii) SMOTE tends to create a large no. of noisy data points in feature space.

2. **ADASYN: Adaptive Synthetic Sampling Approach**

ADASYN is a generalized form of the SMOTE algorithm. This algorithm also aims to oversample the minority class by generating synthetic instances for it. But the difference here is it considers the density distribution, r_i which decides the no. of synthetic instances generated for samples which difficult to learn. Due to this, it helps in adaptively changing the decision boundaries based on the samples difficult to learn. This is the major difference compared to SMOTE.

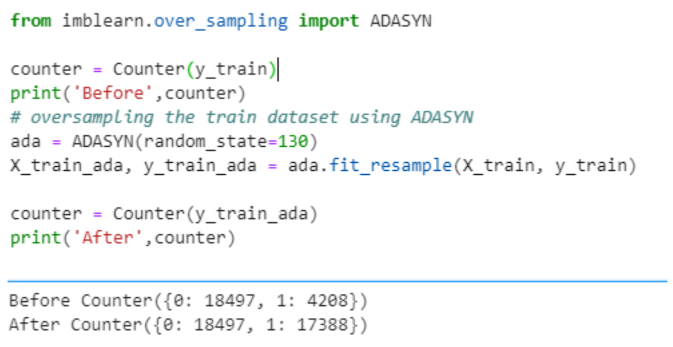
Working Procedure:

- From the dataset, the total no. of majority $\mathbf{N^-}$ and minority $\mathbf{N^+}$ are captured respectively. Then we preset the threshold value, $\mathbf{d^{th}}$ for the maximum degree of class imbalance. Total no. of synthetic samples to be generated, $G = (N^- - N^+) \times \beta$. *Here,* $\boldsymbol{\beta} = \mathbf{(N^+/ N^-)}$.
- For every minority sample x_i, KNN's are obtained using Euclidean distance, and ratio r_i is calculated as $\Delta i/k$ and further normalized as $\mathbf{r_x} <= \mathbf{r_i} / \sum \mathbf{r_i}$.
- Thereafter, the total synthetic samples for each $\mathbf{x_i}$ will be, $\mathbf{g_i} = \mathbf{r_x} \times \boldsymbol{G}$. Now we iterate from ***1*** *to* $\mathbf{g_i}$ to generate samples the same way as we did in SMOTE.

The below-given diagram represents the above procedure:

Source: https://github.com/minoue-xx/Oversampling-Imbalanced-Data

Python code for ADASYN algorithm:



```
from imblearn.over_sampling import ADASYN

counter = Counter(y_train)
print('Before',counter)
# oversampling the train dataset using ADASYN
ada = ADASYN(random_state=130)
X_train_ada, y_train_ada = ada.fit_resample(X_train, y_train)

counter = Counter(y_train_ada)
print('After',counter)
```

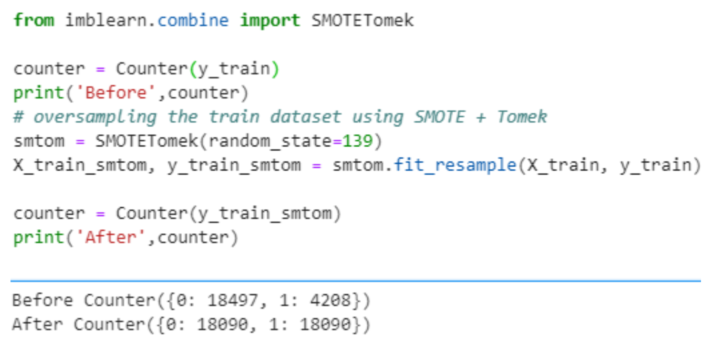
```
Before Counter({0: 18497, 1: 4208})
After Counter({0: 18497, 1: 17388})
```

3. **Hybridization: SMOTE + Tomek Links**

Hybridization techniques involve combining both undersampling and oversampling techniques. This is done to optimize the performance of classifier models for the samples created as part of these techniques.

SMOTE+TOMEK is such a hybrid technique that aims to clean overlapping data points for each of the classes distributed in sample space. After the oversampling is done by SMOTE, the class clusters may be invading each other’s space. As a result, the classifier model will be overfitting. Now, Tomek links are the opposite class paired samples which are closest neighbors to each other. Therefore the majority class observations from these links are removed as it is believed to increase the class separation near the decision boundaries. Now, to get better class clusters, Tomek links are applied to oversampled minority class samples done by SMOTE. Thus instead of removing the observations only from the majority class, we generally remove both the class observations from the Tomek links.

Python code for SMOTE + Tomek algorithm:



```
from imblearn.combine import SMOTETomek

counter = Counter(y_train)
print('Before',counter)
# oversampling the train dataset using SMOTE + Tomek
smtom = SMOTETomek(random_state=139)
X_train_smtom, y_train_smtom = smtom.fit_resample(X_train, y_train)

counter = Counter(y_train_smtom)
print('After',counter)
```

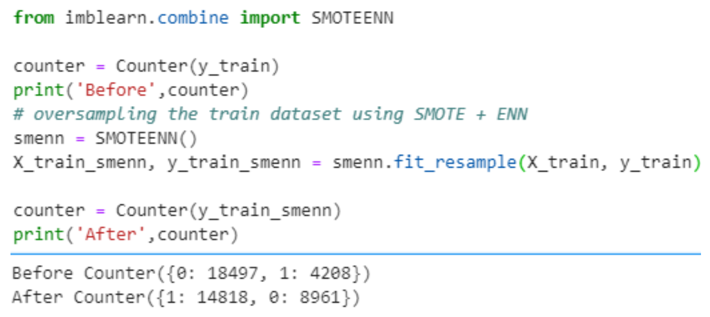
```
Before Counter({0: 18497, 1: 4208})
After Counter({0: 18090, 1: 18090})
```

4. **Hybridization: SMOTE + ENN**

SMOTE + ENN is another hybrid technique where more no. of observations are removed from the sample space. Here, ENN is yet another undersampling technique where the nearest neighbors of each of the majority class is estimated. If the nearest neighbors misclassify that particular instance of majority class, then that instance gets deleted.

Integrating this technique with oversampled data done by SMOTE helps in doing extensive data cleaning. Here on misclassification by NN's samples from both the classes are removed. This results in a more clear and concise class separation.

Python Code for SMOTE + ENN algorithm:



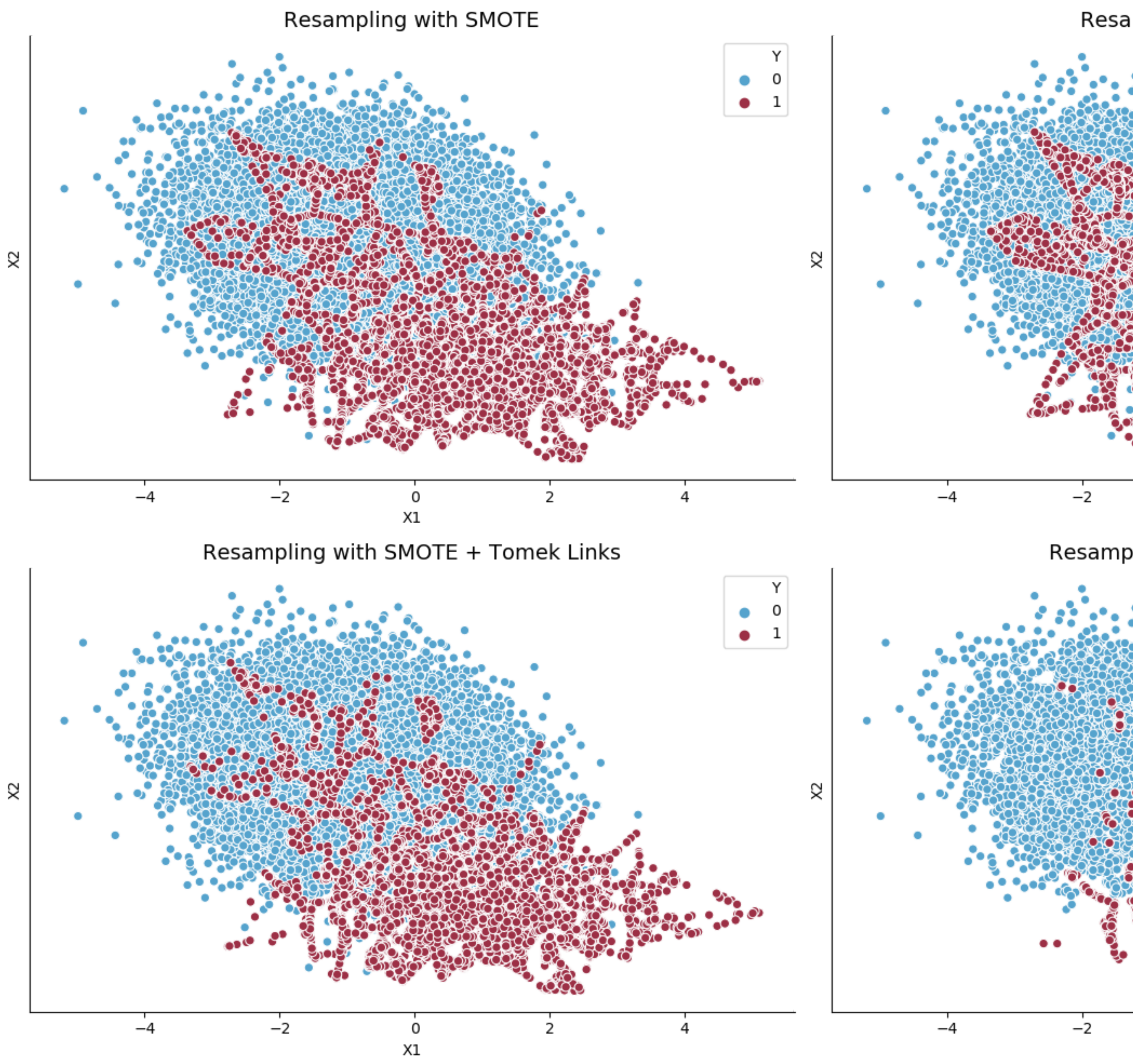
```
from imblearn.combine import SMOTEENN

counter = Counter(y_train)
print('Before',counter)
# oversampling the train dataset using SMOTE + ENN
smenn = SMOTEENN()
X_train_smenn, y_train_smenn = smenn.fit_resample(X_train, y_train)

counter = Counter(y_train_smenn)
print('After',counter)
```

```
Before Counter({0: 18497, 1: 4208})
After Counter({1: 14818, 0: 8961})
```

The below-given picture shows how different SMOTE based resampling techniques work out to deal with imbalanced data.

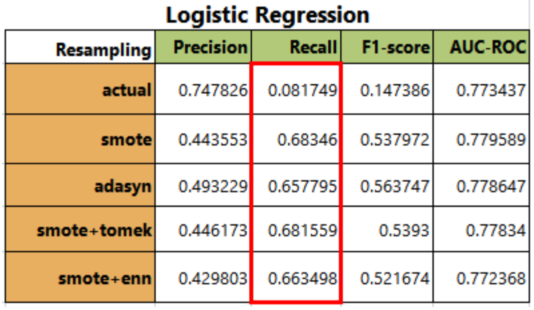


Performance Analysis after Resampling

To understand the effect of oversampling, I will be using a bank customer churn dataset. It is an imbalanced data where the target variable, *churn* has 81.5% customers not churning and 18.5% customers who have churned.

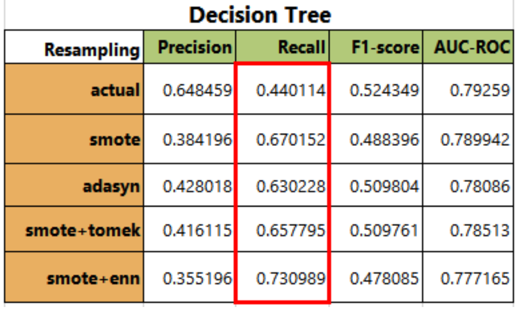
A comparative analysis was done on the dataset using 3 classifier models: Logistic Regression, Decision Tree, and Random Forest. As discussed earlier, we'll ignore the accuracy metric to evaluate the performance of the classifier on this imbalanced dataset. Here, we are more interested to know that which are the customers who'll churn out in the coming months. Thereby, we'll focus on metrics like precision, recall, F1-score to understand the performance of the classifiers for correctly determining which customers will churn.

Note: *The SMOTE and its related techniques are only applied to the training dataset so that we fit our algorithm properly on the data. The test data remains unchanged so that it correctly represents the original data.*



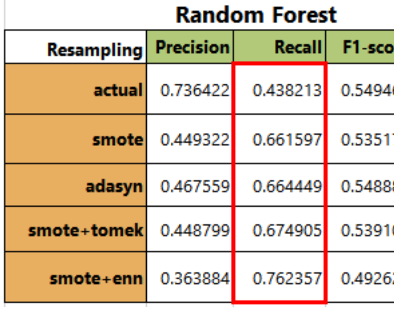
Logistic Regression

Resampling	Precision	Recall	F1-score	AUC-ROC
actual	0.747826	0.081749	0.147386	0.773437
smote	0.443553	0.68346	0.537972	0.779589
adasyn	0.493229	0.657795	0.563747	0.778647
smote+tomek	0.446173	0.681559	0.5393	0.77834
smote+enn	0.429803	0.663498	0.521674	0.772368



Decision Tree

Resampling	Precision	Recall	F1-score	AUC-ROC
actual	0.648459	0.440114	0.524349	0.79259
smote	0.384196	0.670152	0.488396	0.789942
adasyn	0.428018	0.630228	0.509804	0.78086
smote+tomek	0.416115	0.657795	0.509761	0.78513
smote+enn	0.355196	0.730989	0.478085	0.777165



Random Forest

Resampling	Precision	Recall	F1-sco
actual	0.736422	0.438213	0.5494
smote	0.449322	0.661597	0.5351
adasyn	0.467559	0.664449	0.5488
smote+tomek	0.448799	0.674905	0.5391
smote+enn	0.363884	0.762357	0.4926

From the above, it can be seen on the actual imbalanced dataset, all 3 classifier models were not able to generalize well on the minority class compared to the majority class. As a result, most of the negative class samples were correctly classified. Due to this, there was **less FP** compared to **more FN**. After oversampling, a clear surge in **Recall** is seen on the test data. To understand this better, a comparative barplot is shown below for all 3 models:

There is a decrease in precision, but by achieving a much recall which satisfies the objective of any binary classification problem. Also, the AUC-ROC and F1-score for each model remain more or less the same.

End Notes

The issue of class imbalance is just not limited to binary classification problems, multi-class classification problems equally suffer with it. Therefore, it is important to apply resampling techniques to such data so as the models perform to their best and give most of the accurate predictions.

You can check the entire implementation in my GitHub repository and try to apply them at your end. Do explore other techniques that help in handling an imbalanced dataset.

References:

Learning from Imbalanced Data Sets by Alberto Fernández, Salvador García, Mikel Galar, Ronaldo C. Prati, Bartosz Krawczyk, Francisco Herrera

Happy95