ML | Handling Imbalanced Data with SMOTE and Near Miss Algorithm in Python

In Machine Learning and Data Science we often come across a term called **Imbalanced Data Distribution**, generally happens when observations in one of the class are much higher or lower than the other classes. As Machine Learning algorithms tend to increase accuracy by reducing the error, they do not consider the class distribution. This problem is prevalent in examples such as **Fraud Detection**, **Anomaly Detection**, **Facial recognition** etc. Standard ML techniques such as Decision Tree and Logistic Regression have a bias towards the **majority** class, and they tend to ignore the minority class. They tend only to predict the majority class, hence, having major misclassification of the minority class in comparison with the majority class. In more technical words, if we have imbalanced data distribution in our dataset then our model becomes more prone to the case when minority class has negligible or very lesser **recall**. **Imbalanced Data Handling Techniques:** There are mainly 2 mainly algorithms that are widely used for handling imbalanced class distribution.

1. SMOTE
2. Near Miss Algorithm

SMOTE (Synthetic Minority Oversampling Technique) – Oversampling

SMOTE (synthetic minority oversampling technique) is one of the most commonly used oversampling methods to solve the imbalance problem. It aims to balance class distribution by randomly increasing minority class examples by replicating them. SMOTE synthesises new minority instances between existing minority instances. It generates the **virtual training records by linear interpolation** for the minority class. These synthetic training records are generated by randomly selecting one or more of the k-nearest neighbors for each example in the minority class. After the oversampling process, the data is reconstructed and several classification models can be applied for the processed data. **More Deep Insights of how SMOTE Algorithm work !**

- **Step 1:** Setting the minority class set **A**, for each **[Tex]\(x \\in A\)[/Tex]**, the **k-nearest neighbors of x** are obtained by calculating the **Euclidean distance** between **x** and every other sample in set **A**.
- **Step 2:** The sampling rate **N** is set according to the imbalanced proportion. For each **[Tex]\(x \\in A\)[/Tex]**, **N** examples (i.e x1, x2, …xn) are randomly selected from its k-nearest neighbors, and they construct the set [Tex]\(A_1\)[/Tex] .
- **Step 3:** For each example **[Tex]\(x_k \\in A_1\)[/Tex]** (k=1, 2, 3…N), the following formula is used to generate a new example: [Tex]\(x’ = x + rand(0, 1) * \\mid x – x_k \\mid\)[/Tex] in which

rand(0, 1) represents the random number between 0 and 1.

NearMiss Algorithm – Undersampling

NearMiss is an under-sampling technique. It aims to balance class distribution by randomly eliminating majority class examples. When instances of two different classes are very close to each other, we remove the instances of the majority class to increase the spaces between the two classes. This helps in the classification process. To prevent problem of **information loss** in most under-sampling techniques, **near-neighbor** methods are widely used. **The basic intuition about the working of near-neighbor methods is as follows:**

- **Step 1:** The method first finds the distances between all instances of the majority class and the instances of the minority class. Here, majority class is to be under-sampled.
- **Step 2:** Then, **n** instances of the majority class that have the smallest distances to those in the minority class are selected.
- **Step 3:** If there are k instances in the minority class, the nearest method will result in **k*n** instances of the majority class.

**For finding n closest instances in the majority class, there are several variations of applying NearMiss Algorithm 😗 *

1. **NearMiss – Version 1 😗 * It selects samples of the majority class for which average distances to the k **closest** instances of the minority class is smallest.
2. **NearMiss – Version 2 😗 * It selects samples of the majority class for which average distances to the k **farthest** instances of the minority class is smallest.
3. **NearMiss – Version 3 😗 * It works in 2 steps. Firstly, for each minority class instance, their **M nearest-neighbors** will be stored. Then finally, the majority class instances are selected for which the average distance to the N nearest-neighbors is the largest.

This article helps in better understanding and hands-on practice on how to choose best between different imbalanced data handling techniques.

Load libraries and data file

The dataset consists of transactions made by credit cards. This dataset has **492 fraud transactions out of 284, 807 transactions**. That makes it highly unbalanced, the positive class (frauds) account for 0.172% of all transactions.

```
data[``'normAmount'``]  =
StandardScaler().fit_transform(np.array(data[``'Amount'``]).reshape(``-``1``,  1``))

data  =  data.drop([``'Time'``,  'Amount'``], axis  =  1``)
```

```
data[``'Class'``].value_counts()
```

Output:

RangeIndex: 284807 entries, 0 to 284806 Data columns (total 31 columns): Time 284807 non-null float64 V1 284807 non-null float64 V2 284807 non-null float64 V3 284807 non-null float64 V4 284807 non-null float64 V5 284807 non-null float64 V6 284807 non-null float64 V7 284807 non-null float64 V8 284807 non-null float64 V9 284807 non-null float64 V10 284807 non-null float64 V11 284807 non-null float64 V12 284807 non-null float64 V13 284807 non-null float64 V14 284807 non-null float64 V15 284807 non-null float64 V16 284807 non-null float64 V17 284807 non-null float64 V18 284807 non-null float64 V19 284807 non-null float64 V20 284807 non-null float64 V21 284807 non-null float64 V22 284807 non-null float64 V23 284807 non-null float64 V24 284807 non-null float64 V25 284807 non-null float64 V26 284807 non-null float64 V27 284807 non-null float64 V28 284807 non-null float64 Amount 284807 non-null float64 Class 284807 non-null int64

```
from  sklearn.model_selection  import  train_test_split

X_train, X_test, y_train, y_test  =  train_test_split(X, y, test_size  =  0.3``, random_state
=  0``)

print``(``"Number transactions X_train dataset: "``, X_train.shape)

print``(``"Number transactions y_train dataset: "``, y_train.shape)

print``(``"Number transactions X_test dataset: "``, X_test.shape)

print``(``"Number transactions y_test dataset: "``, y_test.shape)
```

Output:

0 284315 1 492

Split the data into test and train sets

```
print``(``"Before OverSampling, counts of label '1': {}"``.``format``(``sum``(y_train  =``=
1``)))

print``(``"Before OverSampling, counts of label '0': {} \n"``.``format``(``sum``(y_train  =``=
0``)))

from  imblearn.over_sampling  import  SMOTE

sm  =  SMOTE(random_state  =  2``)

X_train_res, y_train_res  =  sm.fit_sample(X_train, y_train.ravel())
```

```
print``(``'After OverSampling, the shape of train_X: {}'``.``format``(X_train_res.shape))

print``(``'After OverSampling, the shape of train_y: {} \n'``.``format``(y_train_res.shape))

print``(``"After OverSampling, counts of label '1': {}"``.``format``(``sum``(y_train_res  =``=
1``)))

print``(``"After OverSampling, counts of label '0': {}"``.``format``(``sum``(y_train_res  =``=
0``)))
```

Output:

Number transactions X_train dataset: (199364, 29) Number transactions y_train dataset: (199364, 1) Number transactions X_test dataset: (85443, 29) Number transactions y_test dataset: (85443, 1)

Now train the model without handling the imbalanced class distribution

```
<div  id``=``"highlighter_843442"  class``=``"syntaxhighlighter nogutter  "``><table
border``=``"0"  cellpadding``=``"0"  cellspacing``=``"0"``><tbody><tr><td  class``=``"code"``>
<div  class``=``"container"``><div  class``=``"line number1 index0 alt2"``><code
 class``=``"plain"``>lr1 <``/``code><code  class``=``"keyword"``>``=``<``/``code> <code
 class``=``"plain"``>LogisticRegression() <``/``code><``/``div><div  class``=``"line number2
index1 alt1"``><code  class``=``"plain"``>lr1.fit(X_train_res, y_train_res.ravel()) <``/``code>
<``/``div><div  class``=``"line number3 index2 alt2"``><code  class``=``"plain"``>predictions
<``/``code><code  class``=``"keyword"``>``=``<``/``code> <code
 class``=``"plain"``>lr1.predict(X_test) <``/``code><``/``div><div  class``=``"line number4
index3 alt1"``><code  class``=``"undefined spaces"``> <``/``code> <``/``div><div
 class``=``"line number5 index4 alt2"``><code  class``=``"comments"``>
```

Output:

precision recall f1-score support 0 1.00 1.00 1.00 85296 1 0.88 0.62 0.73 147 accuracy 1.00 85443 macro avg 0.94 0.81 0.86 85443 weighted avg 1.00 1.00 1.00 85443

The accuracy comes out to be 100% but did you notice something strange ? The recall of the minority class in very less. It proves that the model is more biased towards majority class. So, it proves that this is not the best model. Now, we will apply different **imbalanced data handling techniques** and see their accuracy and recall results.

Using SMOTE Algorithm

```
lr2  =  LogisticRegression()

lr2.fit(X_train_miss, y_train_miss.ravel())
```

```
predictions  =  lr2.predict(X_test)

print``(classification_report(y_test, predictions))
```

Output:

Before OverSampling, counts of label '1': [345] Before OverSampling, counts of label '0': [199019] After OverSampling, the shape of train_X: (398038, 29) After OverSampling, the shape of train_y: (398038,) After OverSampling, counts of label '1': 199019 After OverSampling, counts of label '0': 199019

Look! that SMOTE Algorithm has oversampled the minority instances and made it equal to majority class. Both categories have equal amount of records. More specifically, the minority class has been increased to the total number of majority class. Now see the accuracy and recall results after applying SMOTE algorithm (Oversampling).

Prediction and Recall

```
print``(``"Before Undersampling, counts of label '1': {}"``.``format``(``sum``(y_train  =``=
1``)))

print``(``"Before Undersampling, counts of label '0': {} \n"``.``format``(``sum``(y_train
=``=  0``)))

from  imblearn.under_sampling  import  NearMiss

nr  =  NearMiss()

X_train_miss, y_train_miss  =  nr.fit_sample(X_train, y_train.ravel())

print``(``'After Undersampling, the shape of train_X: {}'``.``format``(X_train_miss.shape))

print``(``'After Undersampling, the shape of train_y: {} \n'``.``format``(y_train_miss.shape))

print``(``"After Undersampling, counts of label '1': {}"``.``format``(``sum``(y_train_miss
=``=  1``)))

print``(``"After Undersampling, counts of label '0': {}"``.``format``(``sum``(y_train_miss
=``=  0``)))
```

Output:

precision recall f1-score support 0 1.00 0.98 0.99 85296 1 0.06 0.92 0.11 147 accuracy 0.98 85443 macro avg 0.53 0.95 0.55 85443 weighted avg 1.00 0.98 0.99 85443

Wow, We have reduced the accuracy to 98% as compared to previous model but the recall value of minority class has also improved to 92 %. This is a good model compared to the previous one. Recall is great. Now, we will apply NearMiss technique to Under-sample the majority class and see its accuracy and recall results.

NearMiss Algorithm:

train the model on train set lr2 = LogisticRegression() lr2.fit(X_train_miss, y_train_miss.ravel()) predictions = lr2.predict(X_test) # print classification report print(classification_report(y_test, predictions))

Output:

Before Undersampling, counts of label '1': [345] Before Undersampling, counts of label '0': [199019] After Undersampling, the shape of train_X: (690, 29) After Undersampling, the shape of train_y: (690,) After Undersampling, counts of label '1': 345 After Undersampling, counts of label '0': 345

The **NearMiss Algorithm** has undersampled the majority instances and made it equal to majority class. Here, the majority class has been reduced to the total number of minority class, so that both classes will have equal number of records.

Prediction and Recall

```
<div id``=``"highlighter_764484" class``=``"syntaxhighlighter nogutter "``><table border``=``"0" cellpadding``=``"0" cellspacing``=``"0"``><tbody><tr><td class``=``"code"``><div class``=``"container"``><div class``=``"line number1 index0 alt2"``><code class``=``"comments"``>
```

Output:

precision recall f1-score support 0 1.00 0.56 0.72 85296 1 0.00 0.95 0.01 147 accuracy 0.56 85443 macro avg 0.50 0.75 0.36 85443 weighted avg 1.00 0.56 0.72 85443

This model is better than the first model because it classifies better and also the recall value of minority class is 95 %. But due to undersampling of majority class, its recall has decreased to 56 %. So in this case, SMOTE is giving me a great accuracy and recall, I'll go ahead and use that model! 😳