Data Mining: Classification

BIASED DATASET

What is a Biased Dataset?

Data bias in machine learning is a type of error in which certain elements of a dataset are more heavily weighted and/or represented than others. Also, data does not include variables that properly capture the phenomenon we want to classify. A biased dataset does not accurately represent a data mining model results, resulting in skewed outcomes, low accuracy levels, and analytical errors.

OVERSAMPLING & UNDER-SAMPLING

Imbalanced datasets are those where there is a severe skew in the class distribution, such as 1:100 or 1:1000 examples in the minority class to the majority class.

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 under-sampling, and to duplicate examples from the minority class, called oversampling.
In this note, I will show random oversampling and under-sampling for imbalanced classification. And you will learn:

- Random resampling provides a naive technique for rebalancing the class distribution for an imbalanced dataset.
- Random oversampling duplicates examples from the minority class in the training dataset and can result in overfitting for some models.
- Random under-sampling deletes examples from the majority class and in the training dataset and can result in losing information valuable to a model.

Random Resampling Imbalanced Datasets

Resampling involves creating a new transformed version of the training dataset in which the selected examples have a different class distribution. Resampling:

- is a simple and effective strategy for imbalanced classification problems.
- is the simplest strategy, means simply randomly choosing examples for the transformed dataset .

There are two main approaches to random resampling for imbalanced classification; they are oversampling and under-sampling.

- *Random Oversampling: Randomly duplicate examples in the minority class.*
- *Random Under-sampling: Randomly delete examples in the majority class.*

Random oversampling involves randomly selecting examples from the minority class, with replacement, and adding them to the training dataset. Random under-sampling involves randomly selecting examples from the majority class and deleting them from the training dataset.

In the random under-sampling, the majority class instances are discarded at random until a more balanced distribution is reached

In the random over-sampling, the minority class instances are duplicated at random until a more balance distribution is reached

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 "*naïve 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 validation, test, or holdout dataset used to evaluate the performance of a model.

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

Imbalanced-Learn Library (R VERSION)

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

In the following R imbalanced-learn library we use newthyroid1 dataset which is available in package R.

I used an already existing imbalance dataset (newthyroid1) available in the "imbalance" package.

```
install.packages("imbalance")
library(imbalance)
table(newthyroid1$Class)
head(newthyroid1, 5)
'#
  T3resin Thyroxin Triiodothyronine Thyroidstimulating TSH_value    Class
1     105      7.3              1.5                 1.5      -0.1 negative
2      67     23.3              7.4                 1.8      -0.6 positive
3     111      8.4              1.5                 0.8       1.2 negative
4      89     14.3              4.1                 0.5       0.2 positive
5     105      9.5              1.8                 1.6       3.6 negative
#'
```

You can use imbalanceRatio() to get the imbalance ratio,

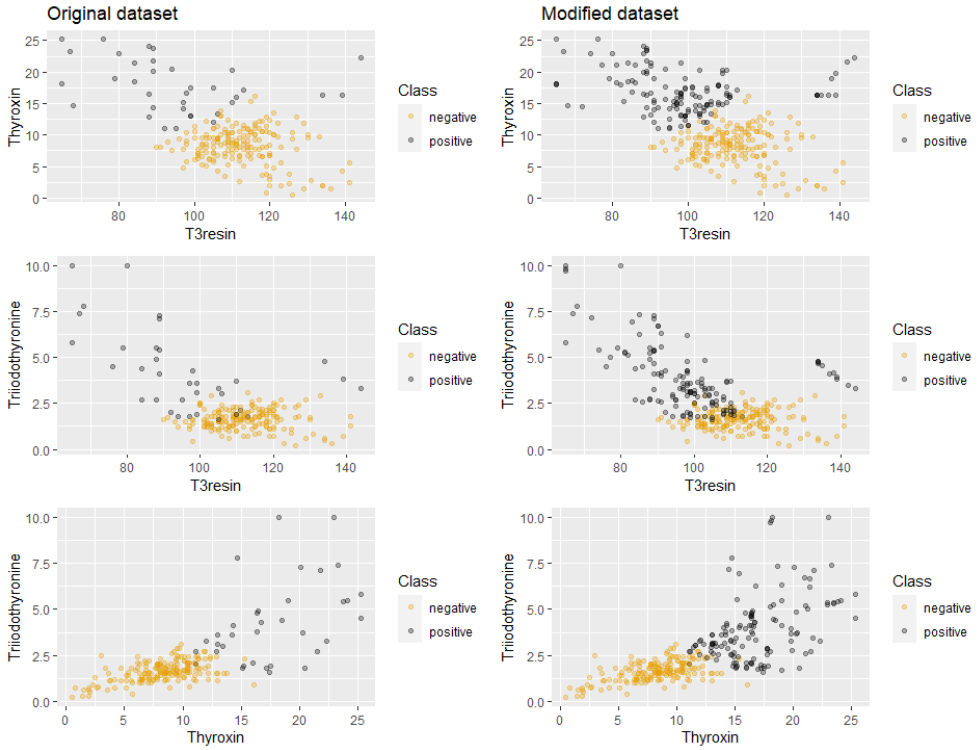
```
imbalanceRatio(newthyroid1)
## [1] 0.1944444
```

The following R (if you are using Python, you may get the same function from Python library) functions provide over or under-sampling datasets which could be added to the biased training dataset, so data mining model is built over an unbiased dataset.

1. MWMOTE

Majority Weighted Minority Oversampling TEchnique (MWMOTE)
MWMOTE is an extension of the original SMOTE algorithm. It assigns higher weight to borderline instances, undersized minority clusters and examples near the borderline of the two classes.
This sample codes, first 100 new observations are created which, their class is the minority class and are similar to those majority borderline class. Then, these new observations are added to the original dataset (training) which now is not imbalanced anymore.

```
#Creating 100 new records of minority class
newMWMOTE <-mwmote(newthyroid1, numInstances = 100)
nrow(newMWMOTE)
head(newMWMOTE)
'#
  T3resin Thyroxin Triiodothyronine Thyroidstimulating     TSH_value    Class
1     106 16.17721         0.4912386          2.0885884  0.0028457337 positive
2     106 19.01538         2.7614889          1.2058718 -0.2030073304 positive
3     111 10.91215         1.1420163          1.0350104  0.1451645526 positive
4     111 19.70969         3.7626941          0.2586344 -0.0005115905 positive
5      95 22.00905         2.2745543          1.0152806  0.0430844863 positive
#'
table(newMWMOTE$Class)
'#
negative positive
       0      100
#'
newMWMOTE.Plus<-rbind(newthyroid1, newMWMOTE)
table(newMWMOTE.Plus$Class)
'#
A more balance dataset
negative positive
     180      135
#'
plotComparison(newthyroid1,newMWMOTE.Plus,attrs = names(newthyroid1)[1:3])
```

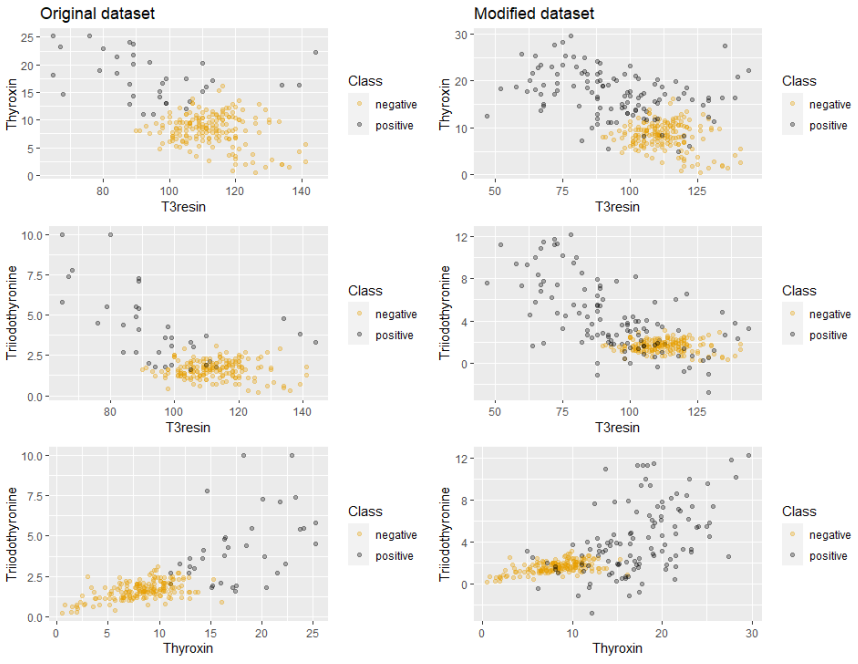


2. PDFOS

Probability Distribution density Function estimation based OverSampling
PDFOS uses multivariate Gaussian kernel methods to locally approximate the minority class.

We run pdfos (Probabiity Density Function) algorithm on newthyroid1 imbalanced dataset and plot a comparison between attributes. See https://rdrr.io/cran/imbalance/man/pdfos.html link. First 80 new observation of the minority class is generated. Then, it is combined with the original dataset with imbalanced observations

```
newSamples <- pdfos(newthyroid1, numInstances = 80)
newDataset <- rbind(newthyroid1, newSamples)
imbalanceRatio(newDataset)
## [1] 0.6388889

newpdfos <- pdfos(newthyroid1, numInstances = 80)
nrow(newpdfos)
head(newpdfos)
'#
  T3resin Thyroxin Triiodothyronine Thyroidstimulating     TSH_value    Class
1     106 16.17721        0.4912386          2.0885884  0.0028457337 positive
2     106 19.01538        2.7614889          1.2058718 -0.2030073304 positive
3     111 10.91215        1.1420163          1.0350104  0.1451645526 positive
4     111 19.70969        3.7626941          0.2586344 -0.0005115905 positive
5      95 22.00905        2.2745543          1.0152806  0.0430844863 positive
6      64 20.61920        8.0138068          1.8756941  0.0767086504 positive
#'
table(newpdfos$Class)
'#
A more balanced dataset
negative positive
       0       80
#'
newpdfos.plus <- rbind(newthyroid1, newpdfos)
table(newpdfos.plus$Class)
'#
negative positive
     180      115
#'
```



3. NEATER

Apart from oversampling methods, NEATER is a filtering method. It discards the instances with higher probability of belonging to the opposite class, based on each instance neighborhood.

```
#we use the minority dataset which was built inPDFOS. This filtering create is for
optimization

newNEATER <- neater(newthyroid1, newPDFOS, iterations = 500)
#[1] "8 samples filtered by NEATER"

#then we create our balanced dataset

newNEATER.plus <- rbind(newthyroid1, newNEATER)
plotComparison(newthyroid1, newNEATER.plus, attrs = names(newthyroid1)[1:3])
```

4. oversample

imbalance includes the method oversample, which is a wrapper that eases calls to the described and already existing methods. Possible methods are: RACOG, wRACOG, PDFOS, RWO, ADASYN, ANSMOTE, BLSMOTE, DBSMOTE, BLSMOTE, DBSMOTE, SLMOTE, RSLSMOTE.
Oversample is an integrated method. Here we do not need to combine the original dataset with the new observations of the minority class.

```
filtered_OS<-oversample(newthyroid1, ratio = 1, method = "PDFOS", filtering = TRUE, i
terations = 500 )
```

How to interpret the oversampling dataset

Problem 5.5 textbook.
A large number of insurance records are to be examined to develop a model for predicting fraudulent claims. Of the claims in the historical database, 1% were judged to be fraudulent. A sample is taken to develop a model, and oversampling is used to provide a balanced sample in light of the very low response rate. When applied to this sample (n = 800), the model ends up correctly classifying 310 frauds, and 270 non-frauds. It missed 90 frauds, and classified 130 records incorrectly as frauds when they were not.

Question 1.
Produce the confusion matrix for the sample as it stands.
Classification confusion matrix

		Predicted Class	
		1	0
Actual Class	1	310	90
	0	130	270

Total of 1 = 310 + 90 = 400
Total of 0 = 130 + 270 = 400
miscl.rate1 <- (90 + 130) / 800 = 0.275 or 27.5%
The model ends up classifying
(310 + 130) / 800 = 0.55 = 55% of the records as fraudulent (1).

Question 2.
Find the adjusted misclassification rate (adjusting for the oversampling).

Now we need to add enough zeros so that the 1's only constitute 1% of the total and the 0's constitute 99% of the total (where is X is the total).
So, 400 fraudulent observations is equal to total size of the dataset minus the 99% of the observations which is
400 fraudulent = size of the dataset - 99% size of the dataset
Therefore dataset = 40, 000

Or we can say if 1% of the dataset is 400 then what is the size of the dataset. Which is 400/0.01 = 40,000

And
Number of zeros = 0.99 * 40, 000 = 39600
Number of ones = 0.01 * 40,000 = 400

Now we can update the confusion matrix based on the above explanation
If 130 out of 400 are misclassified as fraudulent (1) then, how many should be from 99% of the dataset.
That is 12870 = (39600 * 130)/400
If 270 out of 400 are correctly classified as non-fraudulent then, how many should be from 99% of the dataset.
That is 26730 = (39600 * 270)/400
And updated confusion matrix is

		Predicted Class	
		1	0
Actual Class	1	310	90
	0	12870	26730

Recalculating
Misclassification rate = (90 + 12870) / 40000 = 0.324 = 32.4%
miscl.rate2 <- (90 + 12870) / 40000
miscl.rate2 0.324
The model ends up classifying (310 + 12870) / 40, 000 = 0.3295 = 32.95% of the records as fraudulent.

5.5.c. What percentage of new records would you expect to be classified as fraudulent? From the above calculations, we expect 32.95% of the records to be classified as frauds.

PRACTICAL GUIDE TO DEAL WITH IMBALANCED CLASSIFICATION PROBLEMS IN R

Introduction

We have several machine learning (ML) algorithms at our disposal for model building. Doing data based prediction is now easier like never before. Whether it is a regression or classification problem, one can effortlessly achieve a reasonably high accuracy using a suitable algorithm. But, this is not the case everytime. Classification problems can sometimes get a bit tricky.

ML algorithms tend to tremble when faced with imbalanced classification data sets. Additionally, they result in biased predictions and misleading accuracies. But, why does it happen ? What factors deteriorate their performance ?

The answer is simple. With imbalanced data sets, an algorithm doesn't get the necessary information about the minority class to make an accurate prediction. For this reason, it is desirable to use ML algorithms with balanced data sets. Then, how should we deal with imbalanced data sets ? The methods are simple but tricky as described in this article.

In this paper, I've shared the important things you need to know to tackle imbalanced classification problems. In particular, I've kept my focus on imbalance in binary classification problems (just two classes). As I try all the time, I've kept the explanation simple and useful. Towards the end, I've provided a practical view of dealing with such data sets in R with ROSE package.

What is Imbalanced Classification ?

Imbalanced classification is a supervised learning problem where one class outnumbers other class by a large proportion. This problem is faced more frequently in binary classification problems than multi-level classification problems.

The term imbalanced refer to the disparity encountered in the dependent (response or target) variable. Therefore, an imbalanced classification problem is one in which the dependent variable has imbalanced proportion of classes. In other words, a data set that exhibits an unequal distribution between its classes is considered to be imbalanced.

For example: Consider a data set with 200,000 observations. This data set consist of customers who received a promotion brochure for certain products. The dependent variable represents whether a customer has been responded to the offers (1) or not (0). After analyzing the data, it was found ~ 98% did not responded and only ~ 2% got responded. This is a perfect case of imbalanced classification.

In real life, does such situations arise more ? Yes! For better understanding, here are some real life examples. Please note that the degree of imbalance varies per situations:

An automated inspection machine which detect products coming off manufacturing assembly line may find number of defective products significantly lower than non- defective products.

- A test done to detect cancer in residents of a chosen area may find the number of cancer affected people significantly less than unaffected people.
- In credit card fraud detection, fraudulent transactions will be much lower than legitimate transactions.
- A manufacturing operating under six sigma principle may encounter 10 in a million defected products.

There are many more real life situations which result in imbalanced data set. Now you see, the chances of obtaining an imbalanced data is quite high. Hence, it's important to learn to deal with such problems for every analyst.

Why do standard ML algorithms struggle with accuracy on imbalanced data?

The methods are widely known as "Sampling Methods". Generally, these methods aim to modify an imbalanced data into balanced distribution using some mechanism. The modification occurs by altering the size of original data set and provide the same proportion of balance.

These methods have acquired higher importance after many researches have proved that balanced data results in improved overall classification performance compared to an imbalanced data set. Hence, it's important to learn them. Below are the methods used to treat imbalanced datasets:

- Under-sampling
- Oversampling
- Synthetic Data Generation
- Cost Sensitive Learning
- Let's understand them one by one

1. Under-sampling

This method works with majority class. It reduces the number of observations from majority class to make the data set balanced. This method is best to use when the data set is huge and reducing the number of training samples helps to improve run time and storage troubles. Under-sampling methods are of 2 types: Random and Informative.

Random under-sampling method randomly chooses observations from majority class which are eliminated until the data set gets balanced. Informative under-sampling follows a pre-specified selection criterion to remove the observations from majority class.

Within informative under-sampling, EasyEnsemble and BalanceCascade algorithms are known to produce good results. These algorithms are easy to understand and straightforward to implement.

EasyEnsemble: At first, it extracts several subsets of independent samples (with replacement) from majority class. Then, it develops multiple classifiers based on combination of each subset with minority class. As you see, it works just like a unsupervised learning algorithm.

BalanceCascade: It takes a supervised learning approach where it develops an ensemble of classifier and systematically selects which majority class to ensemble.

Do you see any problem with under-sampling methods? Apparently, removing observations may cause the training data to lose important information pertaining to majority class.

2. Oversampling

This method works with minority class. It replicates the observations from minority class to balance the data. It is also known as upsampling. Similar to under-sampling, this method also can be divided into two types: Random Oversampling and Informative Oversampling.

Random oversampling balances the data by randomly oversampling the minority class. Informative oversampling uses a pre-specified criterion and synthetically generates minority class observations.

An advantage of using this method is that it leads to no information loss. The disadvantage of using this method is that, since oversampling simply adds replicated observations in original data set, it ends up adding multiple observations of several types, thus leading to overfitting. *Although, the training accuracy of such data set will be high, but the accuracy on unseen data will be worse.*

3. Synthetic Data Generation

In simple words, instead of replicating and adding the observations from the minority class, it overcome imbalances by generates artificial data. It is also a type of oversampling technique.

In regards to synthetic data generation, Synthetic Minority Oversampling TEchnique (SMOTE) is a powerful and widely used method. SMOTE algorithm creates artificial data based on feature space (rather than data space) similarities from minority samples. We can also say, it generates a random set of minority class observations to shift the classifier learning bias towards minority class.

To generate artificial data, it uses bootstrapping and k-nearest neighbors. Precisely, it works this way:

1. Take the difference between the feature vector (sample) under consideration and its nearest neighbor.
2. Multiply this difference by a random number between 0 and 1
3. Add it to the feature vector under consideration
4. This causes the selection of a random point along the line segment between two specific features

R has a very well defined package which incorporates this techniques. We'll look at it in practical section below.

4. Cost Sensitive Learning (CSL)

It is another commonly used method to handle classification problems with imbalanced data. It's an interesting method. In simple words, this method evaluates the cost associated with misclassifying observations.

It does not create balanced data distribution. Instead, it highlights the imbalanced learning problem by using cost matrices which describes the cost for misclassification in a particular scenario. Recent researches have shown that cost sensitive learning have many a times outperformed sampling methods. Therefore, this method provides likely alternative to sampling methods.

I am going to explain this method with an example: A data set of loan applicants is given. We are interested to know if an applicant defaults. The data set contains all the necessary information. An applicant who defaults is labeled as positive class (yes or 1). And, a applicant who pays labeled as negative class (no or 0). The problem is to identify which class an applicant belongs to. Now, let's understand the cost matrix.

There is no cost associated with identifying an applicant who default as positive and an applicant who pay as negative. Right ? But, the cost associated with identifying an applicant who default as negative (False Negative) is much more dangerous than identifying an applicant who pays as positive (False Positive).

Cost Matrix is similar of confusion matrix. But, we are more concerned about false positives and false negatives (shown below). There is no cost penalty associated with True Positive and True Negatives as they are correctly identified.

Actual	Predicted		
		Positive	Negative
	Positive	0	C(FN)
	Negative	C(FP)	0

The goal of this method is to choose a classifier with lowest total cost.

```
Total Cost = C(FN) x FN + C(FP) x FP
```

where,

1. FN is the number of positive observations wrongly predicted
2. FP is the number of negative examples wrongly predicted
3. C(FN) and C(FP) corresponds to the costs associated with False Negative and False Positive respectively. Remember, C(FN) > C(FP).

5. Which performance metrics to use to evaluate accuracy ?

Choosing a performance metric is a critical aspect of working with imbalanced data. Most classification algorithms calculate accuracy based on the percentage of observations correctly classified. With imbalanced data, the results are high deceiving since minority classes hold minimum effect on overall accuracy.

Actual	Predicted		
		Positive	Negative
	Positive	True Positive (TP)	False Negative (FN)
	Negative	False Positive (FP)	True Negative (TN)

The difference between confusion matrix and cost matrix is that, cost matrix provides information only about the misclassification cost, whereas confusion matrix describes the entire set of possibilities using TP, TN, FP, FN. In a cost matrix, the diagonal elements are zero. The most frequently used metrics are Accuracy & Error Rate.

```
Accuracy: (TP + TN)/(TP+TN+FP+FN)
```

```
Error Rate = 1 - Accuracy = (FP+FN)/(TP+TN+FP+FN)
```

As mentioned above, these metrics may provide deceiving results and are highly sensitive to changes in data. Further, various metrics can be derived from confusion matrix. The resulting metrics provide a better measure to calculate accuracy while working on a imbalanced data set:

Precision: It is a measure of correctness achieved in positive prediction i.e. of observations labeled as positive, how many are actually labeled positive.

```
Precision = TP / (TP + FP)
```

Recall: It is a measure of actual observations which are labeled (predicted) correctly i.e. how many observations of positive class are labeled correctly. It is also known as 'Sensitivity'.

```
Recall = TP / (TP + FN)
```

F measure: It combines precision and recall as a measure of effectiveness of classification in terms of ratio of weighted importance on either recall or precision as determined by β coefficient.

```
F measure = ((1 + β)² × Recall × Precision) / ( β² × Recall + Precision )
```

β is usually taken as 1.

Though, these methods are better than accuracy and error metric, but still ineffective in answering the important questions on classification. For example: precision does not tell us about negative prediction

accuracy. Recall is more interesting in knowing actual positives. This suggest, we can still have a better metric to offer to our accuracy needs.

Fortunately, we have a ROC (Receiver Operating Characteristics) curve to measure the accuracy of a classification prediction. It's the most widely used evaluation metric. ROC Curve is formed by plotting TP rate (Sensitivity) and FP rate (Specificity).

```
Specificity = TN / (TN + FP)
```

```
Sensitivity = TP / (TP + FN)
```

Any point on ROC graph, corresponds to the performance of a single classifier on a given distribution. It is useful because if provides a visual representation of benefits (TP) and costs (FP) of a classification data. The larger the area under ROC curve, higher will be the accuracy.

There may be situations when ROC fails to deliver trustworthy performance. It has few shortcomings such as.

1. It may provide overly optimistic performance results of highly skewed data.
2. It does not provide confidence interval on classifier's performance
3. It fails to infer the significance of different classifier performance.

As alternative methods, we can use other visual representation metrics include PR curve, cost curves as well. Specifically, cost curves are known to possess the ability to describe a classifier's performance over varying misclassification costs and class distributions in a visual format. In more than 90% instances, ROC curve is known to perform quite well.

Imbalanced Classification in R

Till here, we've learnt about some essential theoretical aspects of imbalanced classification. It's time to learn to implement these techniques practically. In R, packages such as ROSE and DMwR helps us to perform sampling strategies quickly. We'll work on a problem of binary classification.

ROSE (Random Over Sampling Examples) package helps us to generate artificial data based on sampling methods and smoothed bootstrap approach. This package has well defined accuracy functions to do the tasks quickly.

Let's get started

```
#set path
> path <- "C:/Users/manish/desktop/Data/March 2016"
#set working directory
> setwd(path)

#install packages
> install.packages("ROSE")
> library(ROSE)
```

The package ROSE comes with an inbuilt imbalanced data set named as hacide. It comprises of two files: hacide.train and hacide.test. Let's load it in R environment:

```
> data(hacide)
> str(hacide.train)
'data.frame': 1000 obs. of 3 variables:
$ cls: Factor w/ 2 levels "0","1": 1 1 1 1 1 1 1 1 1 1 ...
$ x1 : num 0.2008 0.0166 0.2287 0.1264 0.6008 ...
$ x2 : num 0.678 1.5766 -0.5595 -0.0938 -0.2984 ...
```

As you can see, the data set contains 3 variable of 1000 observations. *cls* is the response variable. *x1* and *x2*are dependent variables. Let's check the severity of imbalance in this data set:

```
#check table
table(hacide.train$cls)
  0     1
980    20

#check classes distribution
prop.table(table(hacide.train$cls))
```

```
  0     1
0.98  0.02
```

As we see, this data set contains only 2% of positive cases and 98% of negative cases. This is a severely imbalanced data set. So, how badly can this affect our prediction accuracy ? Let's build a model on this data. I'll be using decision tree algorithm for modeling purpose.

```
> library(rpart)
> treeimb <- rpart(cls ~ ., data = hacide.train)
> pred.treeimb <- predict(treeimb, newdata = hacide.test)
```

Let's check the accuracy of this prediction. To check accuracy, ROSE package has a function names *accuracy.meas*, it computes important metrics such as precision, recall & F measure.

```
> accuracy.meas(hacide.test$cls, pred.treeimb[,2])
   Call:
   accuracy.meas(response = hacide.test$cls, predicted = pred.treeimb[, 2])
   Examples are labelled as positive when predicted is greater than 0.5

   precision: 1.000
   recall: 0.200
   F: 0.167
```

These metrics provide an interesting interpretation. With threshold value as 0.5, Precision = 1 says there are no false positives. Recall = 0.20 is very much low and indicates that we have higher number of false negatives. Threshold values can be altered also. F = 0.167 is also low and suggests weak accuracy of this model.

We'll check the final accuracy of this model using ROC curve. This will give us a clear picture, if this model is worth. Using the function *roc.curve* available in this package:

```
> roc.curve(hacide.test$cls, pred.treeimb[,2], plotit = F)
Area under the curve (AUC): 0.600
```

AUC = 0.60 is a terribly low score. Therefore, it is necessary to balanced data before applying a machine learning algorithm. In this case, the algorithm gets biased toward the majority class and fails to map minority class.

We'll use the sampling techniques and try to improve this prediction accuracy. This package provides a function named *ovun.sample* which enables oversampling, undersampling in one go.

Let's start with oversampling and balance the data.

```
#over sampling
> data_balanced_over <- ovun.sample(cls ~ ., data = hacide.train, method =
"over",N = 1960)$data
> table(data_balanced_over$cls)
0    1
980 980
```

In the code above, method *over* instructs the algorithm to perform over sampling. *N* refers to number of observations in the resulting balanced set. In this case, originally we had 980 negative observations. So, I instructed this line of code to over sample minority class until it reaches 980 and the total data set comprises of 1960 samples.

Similarly, we can perform undersampling as well. Remember, undersampling is done without replacement.

```
> data_balanced_under <- ovun.sample(cls ~ ., data = hacide.train, method =
"under", N = 40, seed = 1)$data
> table(data_balanced_under$cls)
0    1
20  20
```

Now the data set is balanced. But, you see that we've lost significant information from the sample. Let's do both undersampling and oversampling on this imbalanced data. This can be achieved using method = "*both*". In this case, the minority class is oversampled with replacement and majority class is undersampled without replacement.

```
> data_balanced_both <- ovun.sample(cls ~ ., data = hacide.train, method =
"both", p=0.5,                              N=1000, seed = 1)$data
> table(data_balanced_both$cls)
0    1
520 480
```

p refers to the probability of positive class in newly generated sample.

The data generated from oversampling have expected amount of repeated observations. Data generated from undersampling is deprived of important information from the original data. This leads to inaccuracies in the resulting performance. To encounter these issues, ROSE helps us to generate data synthetically as well. The data generated using ROSE is considered to provide better estimate of original data.

```
> data.rose <- ROSE(cls ~ ., data = hacide.train, seed = 1)$data
> table(data.rose$cls)
0    1
520 480
```

This generated data has size equal to the original data set (1000 observations). Now, we've balanced data sets using 4 techniques. Let's compute the model using each data and evaluate its accuracy.

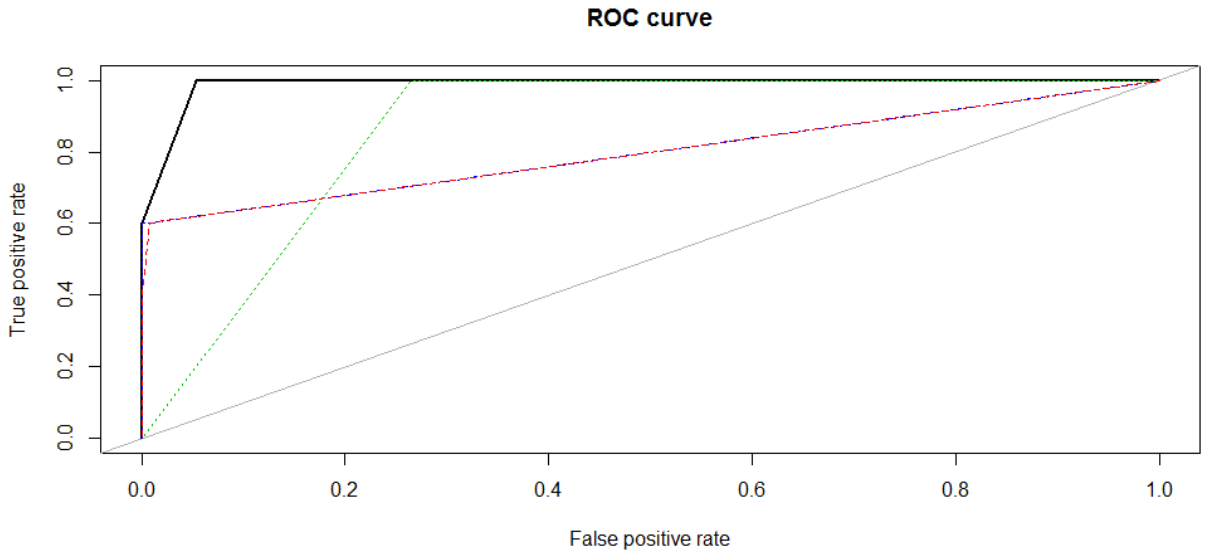
```
#build decision tree models
> tree.rose <- rpart(cls ~ ., data = data.rose)
> tree.over <- rpart(cls ~ ., data = data_balanced_over)
> tree.under <- rpart(cls ~ ., data = data_balanced_under)
> tree.both <- rpart(cls ~ ., data = data_balanced_both)

#make predictions on unseen data
> pred.tree.rose <- predict(tree.rose, newdata = hacide.test)
> pred.tree.over <- predict(tree.over, newdata = hacide.test)
> pred.tree.under <- predict(tree.under, newdata = hacide.test)
> pred.tree.both <- predict(tree.both, newdata = hacide.test)
```

It's time to evaluate the accuracy of respective predictions. Using inbuilt function roc.curve allows us to capture roc metric.

```
#AUC ROSE
> roc.curve(hacide.test$cls, pred.tree.rose[,2])
Area under the curve (AUC): 0.989

#AUC Oversampling
roc.curve(hacide.test$cls, pred.tree.over[,2])
Area under the curve (AUC): 0.798

#AUC Undersampling
roc.curve(hacide.test$cls, pred.tree.under[,2])
Area under the curve (AUC): 0.867

#AUC Both
roc.curve(hacide.test$cls, pred.tree.both[,2])
Area under the curve (AUC): 0.798
```

Here is the resultant ROC curve where:

- Black color represents *ROSE* curve
- Red color represents *oversampling* curve
- Green color represents *undersampling* curve
- Blue color represents *both sampling* curve



Hence, we get the highest accuracy from data obtained using *ROSE* algorithm. We see that the data generated using synthetic methods result in high accuracy as compared to sampling methods. This technique combined with a more robust algorithm (random forest, boosting) can lead to exceptionally high accuracy.

This package also provide us methods to check the model accuracy using holdout and bagging method. This helps us to ensure that our resultant predictions doesn't suffer from high variance.

```
> ROSE.holdout <- ROSE.eval(cls ~ ., data = hacide.train, learner = rpart,
method.assess = "holdout", extr.pred = function(obj)obj[,2], seed = 1)
> ROSE.holdout

Call:
ROSE.eval(formula = cls ~ ., data = hacide.train, learner = rpart,
extr.pred = function(obj) obj[, 2], method.assess = "holdout",
seed = 1)

Holdout estimate of auc: 0.985
```

We see that our accuracy retains at ~ 0.98 and shows that our predictions aren't suffering from high variance. Similarly, you can use bootstrapping by setting method.assess to "BOOT". The parameter extr.pred is a function which extracts the column of probabilities belonging to positive class.